Math

UIUCDCS-R-75-695

AN INTERACTIVE ANALYSIS SYSTEM FOR EXECUTION-TIME ERRORS

by

Alan Mark Davis

January 1975

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

UIUCDCS-R-75-695

AN INTERACTIVE ANALYSIS SYSTEM FOR EXECUTION-TIME ERRORS*

by

Alan Mark Davis

January 1975

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

*This work was supported in part by the National Science Foundation under Grant No. US-NSF-EC-41511 and was submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science, January 1975.

Dedicated to my parents

for their love, support,

and faith which have enabled me

to accomplish this work

in memory of

Morris Slobodow

ACKNOWLEDGEMENTS

The author wishes to express his gratitude to his thesis advisor, Professor Thomas Wilcox, for his very useful guidance and suggestions throughout all stages of this project, and to Professor Jurg Nievergelt for his helpful comments about the implementation and the written thesis.

A sincere thank you must also be given to the other members of the thesis committee: Professors D. B. Gillies, W. Kubitz, and M. D. Mickunas for their active support; as well as to Mike Tindall and Professor W. Hansen for their many helpful suggestions. The National Science Foundation (Grant EC-41511) and the Department of Computer Science of the University of Illinois deserve many thanks for their financial support.

Thanks also goes to the entire PLATO staff and to Professor H. G. Friedman for their assistance in enabling the implementation to be completed. The Fall, 1974 CS 121 class must also be thanked for volunteering to be guinea pigs on the experimental system. Sandy Leach read the manuscript and her multitudinous suggestions on both grammar and content were much appreciated.

Also, a word of thanks is in order for Mary Jane Doolen for her excellent job of typing this manuscript when time was of major importance.

TABLE OF CONTENTS

Page

1. INTRODUCTION

1.1 Purpose

A project is underway at the University of Illinois to automate the beginning (freshman) computer programming courses taught by the Department of Computer Science. The Automated Computer Science Education System (ACSES) hopes to replace most of the work of the human consultants, instructors, teaching assistants and tutors [16]. This project is being implemented on the PLATO computer-assisted instruction system [1]. One essential part of ACSES is a highly interactive programming system on which beginning programming students may write, edit, execute and debug their programs. This Computer-Assisted Programming System (CAPS) will be discussed in Chapter 2 and is described fully in [22].

One of the obvious goals of CAPS is to supply the diagnostic assistance necessary for beginning programmers when either syntax or run-time errors are encountered. Much work has been done in the area of detection, correction, and analysis of syntax errors [12,13,20,21], but research with execution-time errors has been limited to detection and repair methods. Error detection at run-time deals with methods of locating programming bugs as soon as possible and reporting such to the programmer. Error repair at run-time deals with methods of correcting the situation in some way as to permit execution to continue. This thesis is concerned with error analysis. Error analysis at run-time deals with methods of discovering why the error has occurred. Up to now execution-time error analysis has not been explored in depth.

Specifically, the purpose of this thesis is threefold: (1) to present the reasons why analysis of execution-time errors is needed for beginning programmers, (2) to develop the actual features and algorithms necessary for such analysis to be effective, and (3) to describe and analyze an actual implementation of the developed algorithms. It is the purpose of the present chapter to explore the differences between run-time error analysis and the other types of run-time error handling which exist on present programming systems.

1.2 Batch Error Handling

The barest possible execution environment is one in which the intermediate text that has been generated by a compiler is only machine language for the target computer, and the computer simply executes the instructions. In this non-interpretive environment, the only execution errors that can be determined easily are those which the hardware will detect. These would typically include such things as addressing on a non-word boundary, addressing outside the program's data area and arithmetic overflow or underflow. The information given to the programmer would probably include the type of error interrupt triggered and the offset in machine words from the base of the program. The programmer could then, if given the proper cross-reference tables, symbol tables and dumps, locate the statement in which the interrupt occurred.

All of this information is utterly useless to a beginning programmer. An execution environment that could be more useful to the beginner is one in which the intermediate text generated by the compiler is interpreted by an execution supervisor that also monitors the data structures used in order to locate an execution error long before it would

have been detected by the hardware. An interpreter, for example, might be able to flag an array subscript which is out of bounds; however, a non-interpreting execution supervisor might be able to locate it only if the subscripting caused a memory reference outside the user area. Notice that if a program were allowed to modify words which are immediately above or below an array by subscripting out of range, the effects of that error would be disastrous and, if finally detected many statements later, would be extremely difficult to isolate!

Both of the above environments can be visualized by the block chart given in Figure 1.1. The only difference, but a big difference, between them is their effectiveness in locating errors.

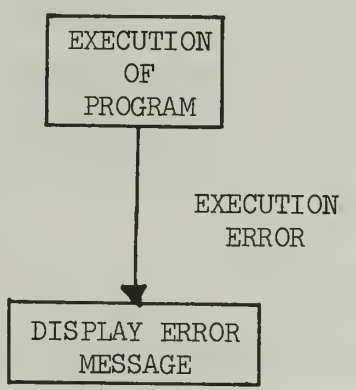


Figure 1.1: Execution Environment Model 1.

The present model of an interpreter can locate only one error at a time during execution. If the interpreter could somehow "repair" the error on the spot as well as supply an error message, then execution could continue until termination or until another error was located. In this case the model looks like Figure 1.2. This technique is used by various batch compilers today [5]. The main problem with them is that neither the error handler nor the execution supervisor know

what the programmer had intended to do. The repair made is quite definitely a "repair" and not a "correction". This repair merely enables execution

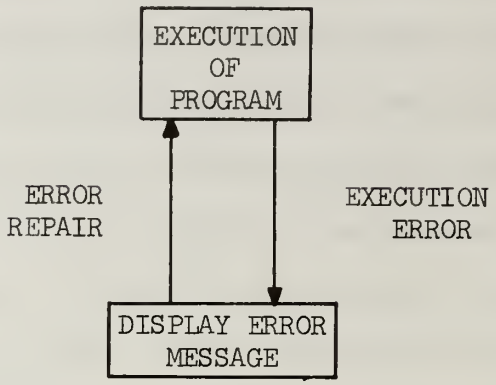


Figure 1.2: Execution Environment Model 2.

to continue unheeded by the presence of an obvious fault in the program. This approach is not a bad one; it is excellent within the confines of a batch environment where there does not exist any method of finding out what the programmer intended.

1.3 Interactive Error Handling

Usually, interactive systems return to the model in Figure 1.1. The main advantage of the interaction is that the programmer is near-by and can therefore interact with a debugging system in order to initiate editing, re-executing, tracing, etc. In this case however, the student is on his own after an execution-time error has occurred. It is felt that this approach lacks necessary guidance by the error system. Many examples of such systems appear in Chapter 2.

This thesis proposes an alternative approach to execution-time errors in an interactive environment. At the occurrence of an error, the basic goal is to discuss with the programmer the problem at hand, obtain from the programmer information about what specific sections of the program

are supposed to do, and finally point out specific changes that the student might make in his program which would prevent the occurrence of the particular error. An additional goal is to perform these tasks in an easy-to-follow orderly manner so as to (1) not confuse a beginning programmer, and (2) demonstrate to the beginning programmer how a program should be debugged in the future by the student himself. This environment is shown in Figure 1.3. This interaction is directed by the system and not by the student. This is essential because the beginning programmer does not know what questions to ask; he does not know how to debug yet. An added result of this is that the student does not have to be burdened with learning another language (i.e., the command language for the debugging package).

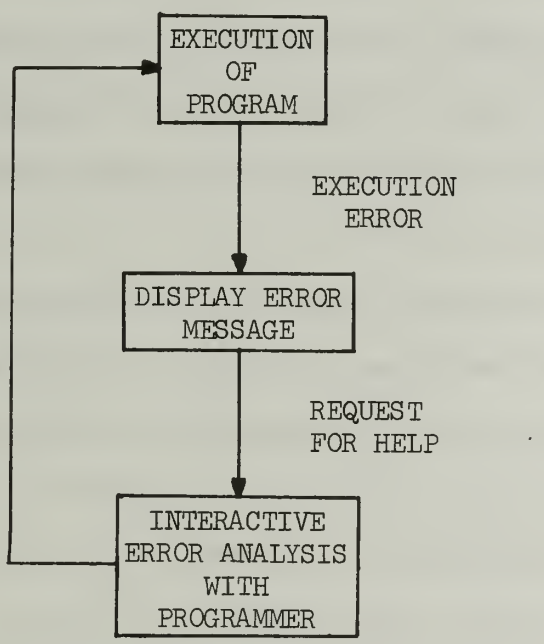


Figure 1.3: Proposed Execution Environment Model 3.

At this time a trivial example should point out the basic differences between the first two models discussed and the proposed model.

Here is a very simple PL/1 program and some sample responses by the three models:

```
line # 1          DECLARE A(1:2);
     # 2          DO I = 0 TO 2 BY 1;
     # 3          A(I) = I;
     # 4          END;
```

```
Model 1:  SUBSCRIPT OUT OF RANGE IN LINE #3.
          EXECUTION HALTED.
```

```
Model 2:  SUBSCRIPT OUT OF RANGE IN LINE #3.
          SUBSCRIPT SET TO 1.
          EXECUTION CONTINUED.
```

```
Model 3:  SUBSCRIPT OUT OF RANGE IN LINE #3.
          DID YOU WANT YOUR ARRAY TO HAVE
             2 ELEMENTS NUMBERED 1 THROUGH 2?
          IF SO, CHANGE LINE #2 TO READ:
                DO I = 1 TO 2 BY 1;
          OTHERWISE, CHANGE LINE #1 TO READ:
                 DECLARE A(0:2);
```

An execution-time error analysis system using this approach has been implemented as part of CAPS and aims at assisting programmers with correcting execution errors as well as teaching students how to debug their programs.

Chapter 2 will discuss the history of debugging systems. Chapter 3 will describe the features of CAPS which influence the run-time error analysis. Chapter 4 will describe the actual run-time error analysis in detail.

2. A SURVEY OF INTERACTIVE DEBUGGING SYSTEMS

2.1 The Past

From the moment that the computer became a useful tool, it became apparent that a major portion of the time used in preparing a program for proper execution on the computer would be spent in debugging. The term debugging means any systematic method of detecting and correcting faulty logic in a program. In this chapter, the reasons for the excessive amount of time spent at debugging will not be explored, nor will new approaches to programming be proposed which could result in fewer "bugs" in the original program. Instead, this chapter will be a survey of the interactive debugging facilities that have been designed to assist the programmer in the burdensome task of eliminating the logic errors from his creation. This survey is intended to point out the shortcomings of the present repertoire of debugging services.

The earliest form of programming was that of assembly language. With each machine and assembly language today the manufacturer usually supplies some form of debugging package. Before these were readily available, the debugger had only the computer's switches (front panel) and perhaps a console to assist him. Through these devices, he was capable of examining or modifying any location in memory and of stepping through his program's execution either by single instructions or processor cycles if necessary. After many attempts at supplying test data and modifying various instructions to comply with his logic constraints, he would

finally debug his program. Debugging in this way was certainly a tiresome task, and luckily there soon arose some new features.

The debugging package afforded the user a welcome set of console commands including setting of breakpoints to permit the program to execute full speed through a debugged section of machine language and then return control to the user at the breakpoint location. Then, the programmer could make requests to see the contents of memory locations, examine registers and status words, and insert or delete additional breakpoints. Because of the very low-level instructions to which the user was restricted, debugging assembly language programs was (and still is) very time-consuming.

As higher level languages became popular, new debugging features had to be developed. No longer should a user be required to know the machine language in order to debug his program. With the advent of more complex operating systems, the user had no way of knowing where his particular program or data resided in memory. Thus there evolved higher level language debugging systems that were applicable in a complex operating system environment which possessed the same features as their assembly language counterparts. However, these debugging systems introduced new implementation problems. For example, it was necessary to have the symbol table present during execution so that the user could refer to data symbolically. Also, it was necessary to locate the beginning and end of source-level statements within all the machine code generated by a typical compiler.

Bernstein and Owens [3] describe four approaches to debugging that could be used:

1. Attempt to duplicate the failing condition in order to obtain information about the program just before failing.
2. Acquire storage maps, tables and data areas.

3. Initiate machine-level debugging procedures at the operators console.
4. Correct the error.

But as time went on, it became apparent that this was not sufficient. In addition to examining and modifying symbolic locations and possibly setting of breakpoints, a few other tools arose which set the trend for debugging systems of today. These included automatic data storage dumps, various trace facilities, and an automatic dump of specific data which described the machine's state at the time of error. Even these tools provide limited information about the cause of errors. The repertoire of debugging tools had to be further extended.

2.2 The Present

What types of services are available today from higher level interactive debugging systems? Project MAC's LISP has a variety of useful debug features [7]. These include trace of both statement flow of control and variable changes, conditional breakpoints, and modification (using an editor) of the actual program's list structures. Conditional breakpoints have a definite advantage over the usual unconditional type. A user may specify in his program that if a certain condition occurs in a certain place, then return control to the user for further requests. The modification to the list structure (both the program and its data are list structures in LISP) is also quite unique. Since this feature is available during interpretation, the programmer may modify the program when control has reached a breakpoint and continue execution; there is no necessity for him to request a recompilation and execution. This is a very useful and timesaving feature, but of course it is easily implemented only in a runtime-storage environment which is as dynamic as that under LISP control.

QUIKTRAN also possesses some unique built-in debugging features [7]. In addition to unconditional breakpoints, examination and modification of variables, trace, and insertion and deletion of FORTRAN statements without a recompilation step, it also has an audit feature. The audit feature, when requested, supplies the programmer with statistics showing the number of times that each statement in his program was executed. This is particularly useful after debugging when the programmer wishes to optimize the most used sections of his program. QUIKTRAN stops just short of a similar feature available in a batch environment described by Wolman [23]. His audit operation supplies the user with the following information about each source level PL/I statement: number of time executed, unit cost (function of amount of machine code generated), total effective cost (number of times executed times unit cost), as well as similar statistics on the time spent on each statement. This is certainly a unique programming aid, but for an experienced programmer!

More conventional debugging features are available on the Berkeley time-sharing system for the MAD and FORTRAN languages [7]. These include acquiring the contents of variables upon request, setting unconditional breakpoints and an extensive editing feature which must be followed by a recompilation. More information about LISP, QUIKTRAN, MADBUG and FORTRAN at Berkeley may be acquired from [14], [6], [8], and [4] respectively.

R. M. Balzer [2] has developed an Extendable Debugging and Monitoring System (EXDAMS) at RAND Corporation with the following goals:

1. To test some proposed but unimplemented debugging facilities,
2. To be extendable to enable new aids to be added easily, and
3. To be independent of the language of the machine.

The basic philosophy of EXDAMS is that in order to perform at an ultimate

level, all a debugging system needs is a complete history of a program's execution. (Contrary to Balzer's opinion, a system requires more than just a program history. This opposing philosophy will be fully discussed in chapter 4.)

Upon the occurrence of an execution error, EXDAMS accepts requests from the programmer for various kinds of information. EXDAMS then looks at the history, extracts the proper information, formats it nicely, and gives it to the programmer for his critical examination. The burden of asking the proper questions (and thus the burden of discovering what caused the error) is regretfully left to the user. However, the features available to assist him in the debugging are quite useful. The programmer has the ability to execute forwards or backwards at any position in the program and may request that these be done at any speed (obviously with some upper limit!) so that the debugger may speed through correct sections of code and may go slowly through possible problem areas. The user may also stop execution at any time and request certain displays. These include "flowback analysis", "control space flowback", and standard tracing facilities. The two flowback features are worth a special note. They are presented to the user when he effectively asks the questions: "How did variable _____ get the value of _____?" and "How did I get from position _____ in the program to position _____?" In the first case, a tree is displayed showing the history of a certain variable as in Figure 2.1. The root of the tree represents the current statement being executed and its descendent nodes represent past events. The debugger may then request more information of this type by indicating in the displayed tree a particular leaf which is to be further expanded into another displayed tree with it at the root. The control space flowback analysis displays a

similar but degenerate (linear) tree that describes the path taken during execution between any two points.

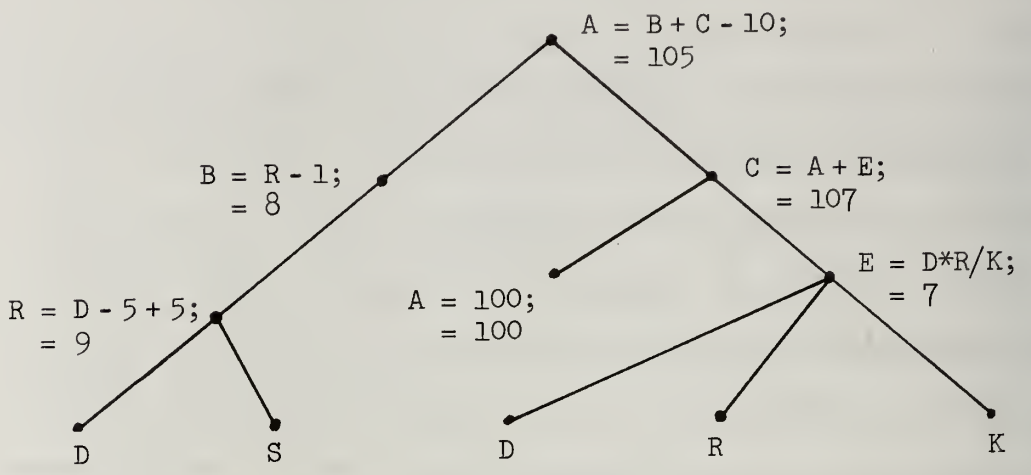


Figure 2.1: EXDAMS' Variable Flowback Analysis

EXDAMS appears to be the most useful and original debugging package of its kind. It combines the more common debugging features with a number of very useful displays; however, its major debugging philosophy appears to agree with most other competitors. That is: First, the user ascertains what has happened; then, if incorrect, the user must determine how or why it happened. This philosophy is obvious when one sees that the user must ask all the questions of the system. This is not an adequate situation because in many cases the programmer has little idea of what went wrong. Programmers are also quite blind to their own bugs; even while staring directly at one, the programmer will not notice a glaring mistake. If the debugging system were somehow "smarter," it could ask the user certain relevant questions about his program and then assist in locating the bug by carefully steering the individual down correct paths of thought.

A new approach to debugging has recently been introduced by Marvin Zelkowitz of Cornell University in his Ph.D. dessertation [24]. He has modified the PL/C compiler to include a modified ON condition statement, a TRACE statement which supplies a statement and variable trace, and a RETRACE statement. The RETRACE statement has the general form,

RETRACE <option> AND <PL/I statement>

and, when executed, will

1. Reverse execute until the <option> is satisfied,
2. Execute the <PL/I statement>, and then
3. Forward execute from the point to which it reversed.

The <option>'s available give the programmer the ability to reverse execute n statements or until any of a set of labels are reached or until a given boolean expression becomes true. In addition, the programmer may specify a list of <option> AND <PL/I statement> pairs separated by OR's with one RETRACE statement. This has the effect of using the first <option> AND <PL/I statement> upon the first encounter of the RETRACE, the second pair upon the second encounter, etc.

This can be a very useful debugging tool even when used as simply as:

```
ON ERROR BEGIN;
      RETRACE  STATEMENTS (50) AND TRACE PRINT;
           OR  CONDITION ('1'B) AND STOP;
      END:
```

which very simply results in a trace of the last 50 statements executed before an error occurred! The obvious advantage of this is that the user does not have to know where his error is before executing. The obvious drawback is the necessity for him to specify the number of statements to

be executed with trace. There is a good possibility that the cause of the error was just a few statements before the error (in which case much needless output has been generated) or that the error was caused by something greater than 50 statements before (in which case the outputted trace is useless). The main cause of this problem is that this particular system is designed for a batch environment. The presence of interaction gives the programmer more freedom of choice. For example, if the user sees the cause of the error right away he could ask to have the trace terminated, otherwise continued. Zelkowitz has made significant progress in the area of introducing new and useful debugging facilities. However, it seems that the effort put into a batch debug system could have been better spent on implementation of the identical features on an interactive system. The added ability of the programmer to intervene in the process would increase its usefulness enormously.

Of course, no discussion of higher level language debugging techniques should neglect to mention the good old-fashioned insertion of PRINT and HALT statements at strategic locations in the program. These statements are practical for debugging purposes only in simple systems where it is very easy to insert and delete source statements (e.g., TUTOR, BASIC). When used properly, they effectively achieve tracing and breakpoint execution.

Another interesting approach was taken by Miller [15] whose system provided for the systematic and automatic generation of test data to test every possible condition that could arise in a program! Other early debugging systems that should also be listed here include Stockham [18], Kulsrud [11] and Jacoby [10].

2.3 The Future (A Critical Analysis of Bernstein and Owens' Ideas for the Future of Debugging Systems.)

Certain suggestions have been set down by Bernstein and Owens [3] for debugging packages of the future. Two of their suggestions are totally reasonable. The tools must be immediately available to the user at his terminal and that they must be conversational in nature. A third suggestion that the tools must be flexible, permitting the user to define his own debugging tools, is good only if applied to an environment with experienced programmers; it renders the tools completely useless to beginning programmers who have no idea of how to debug. The two authors also claim that the debugging system should be completely dormant during normal execution and therefore not affect general program performance. This last statement is true only when applied to a production environment; but, if a loss of performance during the execution of a beginner's program is necessary to supply additional debugging features that he so desperately needs in order to learn about programming and debugging, then the loss is acceptable.

Another statement made by Bernstein and Owens is that debugging facilities should automatically "collect, compare and evaluate data on previous bugs and ultimately ... have the ... [program] ... restructure itself to avoid previously recognized bugs." This is too optimistic. Technology has not yet approached a level that we can consider this realistic. How do they propose to implement the automatic debugging-restructuring program? Their claim is to set it up as a learning program like checkers or chess: "Through extensive interplay of one game program against another, the programs build up statistics and knowledge of the best actions to be taken under each of a multiplicity of conditions."

The problem here is one of the rules of the "games" in question. In chess or checkers, a legal move is well-defined and so is the goal; however, neither of these is particularly easy to state about a program. The implementation would require that the debugging package know exactly what the program's task is. To do this, we need a simple (a programming language specification contains too many hidden bugs) and non-ambiguous (English is much too ambiguous) method of defining a program's goal. Perhaps the solution lies somewhere in the combined efforts of researchers in natural language analysis, artificial intelligence and compiler construction. Section 4.2 describes a method to generate suggestions to the programmer on the basis of accumulated statistics. It does not (and cannot) automatically correct or restructure the program in question.

3. THE PROGRAMMING SYSTEM

3.1 Introduction

The PLATO IV System [1] was designed as a large-scale computer-assisted instruction system at the University of Illinois to support as many as 1000 simultaneous users. Its main processing power comes from two CDC 6400 CPU's augmented by 65K of fast core and one megaword of Extended Core Storage (ECS). The user terminal contains a plasma display panel as its main output component; it was developed at the University of Illinois and is presently being manufactured by Magnavox. PLATO was designed predominantly to facilitate instructional lessons such as question-answer type dialogues. It is quite efficient when performing graphics and text-displaying tasks; however, since a maximum of only 2-3m sec/sec is permitted per user the system cannot be requested to perform extended computing operations.

It is necessary to justify the use of PLATO for the implementation of a programming system that requires considerable processing power. To do this it is necessary to examine the goals of CAPS. The three main goals for the programming system are:

1. Provide a good programming system for the automation of the basic computer science courses at the University of Illinois [16].
2. Design, implement, and test new interactive algorithms for program preparation and execution with error detection, correction, and analysis.

3. Design a completely table-driven system in order to make future language implementation easy.

Since students in the introductory computer science courses will be taught programming language features on PLATO, it is a design goal of ACSES to enable them to write programs on the same system. Thus, PLATO is the logical system on which to build the programming system so far as design goal #1 is concerned. PLATO is also the logical system so far as goal #2 is concerned because PLATO is the only interactive system on campus which is readily accessible. Considering these first two goals, it becomes obvious why PLATO was chosen.

The compiler system on PLATO consists of four logically independent components:

1. The supervisor
2. The editor/compiler package
3. The execution package
4. The filing system.

This document will not discuss components 1 or 4 as they are irrelevant to the thesis material [see 22]. They are mentioned here only for completeness. Discussions of 2 and 3 follow immediately.

3.2 The Editor/Compiler Package

This section is designed to explain to the reader those facets of the editor/compiler which shed light onto the general philosophy of the compiler project or which are relevant to later discussions about execution supervision, error detection and error analysis.

First and foremost, it should be mentioned that the design goal of a completely table-driven compiler has been accomplished. The system

of tables and table drivers has been designed by Wilcox [paper forthcoming]. To implement a new language all that is necessary is to fill in a set of tables. Included among these tables are the LEXI tables which include a classic state-transition table and define for the lexical analyzer the proper groups of characters which are legal tokens in the language. Similarly, the SYNA tables, written by the language implementer in a assembly-like language designed by Tindall [19], define for the syntax analyzer the sequences of tokens which are legal programs and program segments in the source language. As of Autumn of 1974, COBOL, FORTRAN and PL/1 have been implemented on the compiler and their existence demonstrate the flexibility of the compiler tables.

The intermediate text generated by the compiler is very simply a tokenized version of the source text. The advantages of this are that only one copy of the program need be stored, reverse compilation can be implemented easily without complicated reversal of code generation, and the original program can be displayed at any time by examining the intermediate text. The disadvantages are obvious: (1) since there is no universal intermediate text, one execution package cannot execute all languages and (2) the execution supervisors must essentially reparse the entire program and will thus be quite slow.

3.3 The Execution Package

In order to hand tailor an execution supervisor to a beginning programming student, there are two properties which it must have. The first is a trace facility to remove the black box effect. The second is effective error detection and description.

Too often, students who learn on a batch system think that the computer somehow examines the little holes on punched cards and miraculously outputs the correct or incorrect answers. The computer appears as a large expensive black box. It is also interesting to point out that most students associate the term "computer" with the execution phase of a high-level language program. As long as the execution of student's program remains unseen, the "computer" will remain an enigma. CAPS uses a simple, clear, meaningful trace facility. This show the student that an execution supervisor actually executes a program using the same type of logical order that a human programmer uses when he "executes" his program by hand. A statement trace is far more effective at teaching students how a program executes when implemented in an interactive environment than in a batch environment.

Recently, a beginning programming student was testing a program on the PLATO compiler system. After executing her program perhaps six or so times with trace activated, she accidentally requested execution of the program without trace. Upon staring at her apparently static program on the screen for twenty seconds, she exclaimed, "Hey, the computer isn't working." She had obviously been conditioned to <u>seeing</u> the program execute and as soon as she could not see it, she thought that something was wrong. She was right; there is something wrong with batch systems and non-trace systems for small programs which students typically write. Enough said about seeing a program execute.

Since it is impossible to time the statement tracing so that every student is happy with its rate, it is absolutely necessary to provide "faster" students with the ability to increase the speed, and "slower" students with the ability to stop execution at any time and examine the

state of their program. In addition to this, however, it should be permissible for the student to back-up execution at any time in order to see something happen again. This is necessary both for the slower student who misses some action, and for the faster student to gain a fuller understanding of the processes involved. The CAPS execution supervisor provides this feature.

The other area in which an execution supervisor in an interactive student environment must excel is error detection and description. As mentioned in Chapter 1, error detection on the level of that performed by instructional compilers such as PL/C [5] is essential. Error messages should also be clear and to the point. In an interactive environment these can perhaps be stated in a less machine-like fashion as in most compiler systems and instead in a more conversational (but not overly talkative!) manner.

3.4 Language Used For Experimentation

The programming system on PLATO was designed mainly to assist students during the first 6 to 8 weeks of instruction in a new programming language. In light of this, it is necessary to implement only subsets of languages on the system. [22] describes a PL/I subset sufficient for this task. The following is a partial listing of the subset and corresponds to that part which has been implemented for initial use by students. It also corresponds to that subset which was in operation at the time of implementation and experimentation of the execution error analysis system.

1. DECLARE or DCL statement
   - attributes: DECIMAL, DEC, CHARACTER(), CHAR(), VARYING, VAR, FLOAT, FIXED
   - one and two dimensional arrays.

2. DO; END; loops and groups
   - WHILE (expression)
   - index = expression TO expression BY expression WHILE (expression).
3. IF-THEN-ELSE statement.
4. Assignment statement
   - built-in functions: SIN, COS, TAN, LOG, EXP, ABS, MIN, MAX, ATAN, LOG10, FLOOR, CEIL, SUBSTR, INDEX, VERIFY, LENGTH.
5. PROCEDURE OPTIONS (MAIN) statement.
6. GET statement
   - LIST( )
   - SKIP( )
7. PUT statement
   - PAGE( )
   - SKIP( )
   - LIST( )
8. STOP statement.
9. GOTO statement.

For this subset it is necessary to trap the following execution errors at once:

1. Label in GOTO within inactive block,
2. Undefined label,
3. Lower limit greater than upper limit in declaration of an array,
4. Division by zero,
5. Uninitialized variable or array element,
6. Arithmetic overflow/underflow,
7. Argument must be positive,

8. Subscript out of range,

9. Badly formatted input,

10. SUBSTR index less than 1,

11. SUBSTR length less than 0,

12. SUBSTR length + index greater than string length + 1,

as well as any implementation-dependent errors such as

13. Memory limit exceeded,

and any errors resulting from reverse execution such as

14. Reverse execution illegal beyond this position.

4. ANALYSIS OF EXECUTION-TIME ERRORS

4.1 Motivation

Most facets described in the previous chapter can also be found in other existing systems, although perhaps not all in any one. But in all present systems there exists a total lack of assistance to the user after an execution error message has been displayed. LaFrance [12], Levy [13], and Tindall [20, 21] have done excellent jobs of designing algorithms and systems to supply the programmer with suggestions and corrective actions after syntax errors are detected during compilation, but no work has previously been done to offer a student suggestions as to how he may correct an execution error. This chapter explores the features of an interactive system that analyzes the possible causes for an execution error in a program and offers meaningful solutions to the programmer.

Because no software had previously been designed to analyze execution-time errors, the only alternative was to have either the programmer himself or else a human consultant perform the task. Experienced programmers can usually find the causes for their execution errors easily because they possess the knowledge of how to debug a program. Inexperienced programmers typically rely on the consultant to find the cause and to tell the student what to change. His program may work correctly then, but he has not learned anything about how to find the proper correction by himself next time. Ideally, the human consultant should explain the debugging process to the student. However, this is impractical because it would be very time-

consuming to explain this technique to every student, especially when there is a large ratio of students to consultants. Also, knowledge of the types of common misconceptions which students typically have about a language is an important part of being an effective consultant. There is no quick way for the consultant to impart this knowledge to the student; learning them simply takes experience.

It would be nice if software could be designed which (1) could mimic the human consultant and be at least as effective in locating the cause of an execution error, and (2) could perform its tasks in a way so transparent that the student could learn to do the same thing himself. To more fully understand the execution-time error analysis system, it is helpful to examine the following model of a session where a student visits a human consultant.

The session begins with the student telling the consultant that he has received an error at a particular location. When the consultant hears this, he may remember that many students that day have received the same error, and if so, would offer the student some suggestions without even looking at the program in question. This may send the student away to correct his program or it may not be sufficient for his problem. Then the consultant examines the program carefully, looking backwards through the program starting at the location of the execution error. As he notices various potential problem spots, he might ask the student what his intentions were at particular positions. Eventually, he finds a position where there exists a discrepancy between what the code does and what the programmer intended it to do. This is pointed out to the student who would either edit his program or ask for information about how that change could correct the error in question.

Another technique used by a consultant is his overall study of the structure of the program. Since the consultant, with his experience, has some knowledge of the best way to structure the program, he may offer suggestions on how to restructure the program in order to make its logic easier to follow and therefore easier to debug. This cannot be automated because it can be accomplished only by having a knowledge of the problem that the program was written to solve. The assumption will always be made that the error analysis program possesses knowledge only about the debugging of a general program and does not possess knowledge of what the program is supposed to be doing. Thus, with this assumption, there is little means of making global structure suggestions.

In an effort to design software able to analyze programs as well as a human does, the following chart extracts from the previous paragraph exactly what is going on. Namely it shows what the consultant tells the student, what knowledge the consultant must have to do it, what the student tells the consultant and what knowledge the student must possess.

Consultant to Student	Knowledge needed
1. General suggestions for correction without looking at actual program.	1. Memory of previous students' problems and their solutions.
2. After scanning program, questions about particular potential trouble spots.	2. What trouble spots in this program and in this language look like.
3. Information about a discrepency between program and programmer.	3. Ability to find such a condition.
4. Information about how a change in some position will resolve the execution error in question.	4. Ability to show flow of control paths through a program.

Student to Consultant	Knowledge needed
1. What error occurred and where.	1. Ability to read.
2. His intentions at particular positions in his code.	2. What his program is supposed to do and how it is supposed to do it.
3. Whether he understands the correlation between a particular correction and the elimination of an execution error.	3. Ability to talk (or write).

In summary, the consultant must know how to debug and have the experience with students and with the language. The student, as can be seen from the above list, needs little beyond the knowledge of how his program is supposed to do what it was designed to do.

4.2 Introduction

When an execution error occurs it is because the execution supervisor has determined some obvious anomaly in the run-time environment of the program. This error may have been caused by either a problem in the static data structures of a program (e.g., variable attributes) or the dynamic data structures of a program (e.g., variable values). In the former case, the error analysis routines need only examine the static data structures to determine the problem at hand so as to give the student a complete picture of what is going on. In the latter case, it is necessary to examine the dynamic data structures to determine the cause of the error.

Since the static data structures are fixed in time, the examination of them can be made at the time of error with no loss of information. The routines that perform this task are called static analysis routines. Their

goal is (1) to produce a general list of possible causes for the error and (2) to discuss with the student various anomalies in the static run-time data structures. Typically, these would include such things as declarations of various variables but NOT the values contained within the variables.

Since the dynamic data structures are changing in time, a simple examination of them is NOT sufficient to diagnose the cause of the error. It is quite possible that the anomaly found by the execution supervisor is simply a side effect of a bug which is located somewhere earlier in the program. In this case, it is necessary to reverse execute the program in order to locate the first time that an anomaly was present. The routines that perform this task, the dynamic analysis routines, ask questions to the student while it reverse executes in order to find out where the very first anomaly appeared. This position would be the seat of the cause of the execution error. This position could be called the target anomaly of the dynamic analysis system [17].

The following sections of this chapter describe the actual error analysis system that has been designed with the "visit" of section 4.1 as a model and the description in this section as the main philosophy.

4.3 Static Analysis

4.3.1 Introduction

To fully understand this and later chapters, it is necessary to recognize the essential difference between an execution-time error and a cause of that error. An execution-time error is any faulty condition that can arise within the data structures of the program's execution environment which an interpreter or other execution supervisor can discover.

A cause for that error is any typographic, logic, or language usage mistake which, if corrected, would prevent the execution error in question from occurring. Note that this cause may reside at the same location as the execution error or anywhere else in the program.

The general idea behind static analysis is to supply as much information as possible to the student about his error and possible causes for his error with a minimum of computer effort. What is produced is a list of suggestions to the student which may or may not refer to his particular program. They are simply guesses, or intelligent hunches derived from the system's memory of other students making the same error and their successful corrections. It is the decision of the individual student to accept or reject any of these as a correction for his error.

The system's job is similar to the human consultant's when he explains various guesses as to the cause of error by simply using his memory of previous mistakes made during the last few days or weeks. The reasons for both are simple: Human time and computer time are expensive. If a simple comment or two about possible causes could trigger the student's mind to see the bug in his own program, then the costly process of examining details of the particular program can be eliminated. But the consultant relies on his memory to find alternatives. The following sections explore the methods that the error analysis system has to use to remember the same types of alternatives.

4.3.2 What Must be Remembered

To produce a list of meaningful suggestions to the student, it is certainly necessary to have recorded the relationships between execution errors and causes for those errors. Since some systems detect an enormous

number of execution errors, it would be very nice to be able to classify the errors into various types. The criteria for such typing would be to place execution errors with the same list of common causes into the same class.

One obvious example of this is the class of errors caused by some subscript, parameter, argument, or component of an arithmetic expression possessing a zero value illegally. It is certainly probable that the set of reasons that variable B has the value zero in

```
C = A/B;
```

and

```
C = SUBSTR (A,B,D):
```

are not disjoint and in fact might be identical. Although the two errors, "division by zero" and "SUBSTR index less than one", are different they do share the same set of common causes.

When classifying errors by their causes, there are three obvious classifications: (1) those caused by expression components having negative values, (2) those caused by zero values, and (3) those caused by positive values. In fact, most execution errors whose causes lie somewhere other than at the actual location of error do fall into one or more of these three classifications. Notable exceptions to this generalization are errors such as uninitialized variables and undefined labels which would have to fall into other error type groups. The entire purpose for the classification of errors into groups is to reduce the size of the tables which correlate causes of errors with errors.

Ideally, the tables then are two-dimensional structures with each entry (i,j) representing the probability that misconception i is a cause for error class j. But in reality it is not so easy because these tables are not fixed but must change dynamically as students make new

errors. It is quite important, in fact, not to ever convert these into static tables. It is important to continually learn what new mistakes are made as time progresses because the probability that a programming mistake will be the cause of an error will constantly be changing. They will change during a semester as students mature in their programming skills. They will also change in response to a lecturer error or to a detail which a lecturer simply has not made clear to the class. It is necessary in static analysis, as it is with human consultation, to be constantly aware of these fluctuations.

As well as being necessary to adapt automatically to trends in time, it is also necessary to adapt automatically to trends in individual students. If a particular student is slower than most and has difficulty grasping some concepts, then the system should also adapt to these situations. Thus the actual databases used in an implementation must be considerably more complex than simply a two-dimensional array. The actual tables used in the implementation are described in section 4.3.4, and are shown in their entirety in Appendix A.

4.3.3 What Static Analysis Does

As stated in section 4.3.1, when a student encounters an execution error, static analysis routines refer to the databases which have collected information about his tendencies as well as those of other students in his course. By a method to be described later, a list of common causes for this error is generated and displayed to the student. For example, in response to the execution error "SUBSTR index less than 1", PL/I might produce a list similar to that in Figure 4.1.

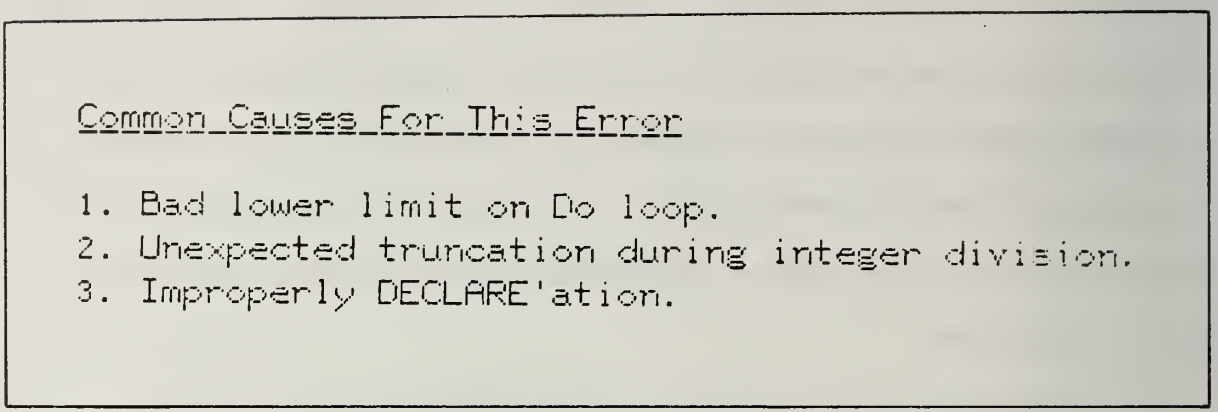


Figure 4.1 Sample Common Cause List

At this time, the student should look at his program and see if any of the suggestions made apply in his case. If so, and if the student sees how to correct his error, then he would simply return to the editor, modify his program, and rerun the execution supervisor. If the list has no effect, then the student would ask for more help.

Associated with each misconception listed in the table, there is a dialogue template. A dialogue template is a skeleton of a discussion with the student about the possible cause of the error and about how the student could modify various statements if his program is to eliminate the execution error.

If the student requests more help, there may be still more information that can be supplied to the student. In particular, it might be known that some of the dialogues associated with the causes listed in the

first part of static analysis are such that they must be relevant to every program which produces a particular error. If this is the case, then certainly those discussions can be initiated at this stage of analysis. As an example of this, let us say that "subscript out of range" is the error in question. One of the common causes displayed might be "improper declaration of the array." It is a fact that a discussion of the possibility of an incorrect declaration is relevant to every program with this error. Thus it would be proper to discuss this matter with the student at this time. A dialogue template for this misconception might look like Figure 4.2. When filled in with the proper information for a particular program, it might look like Figure 4.3.

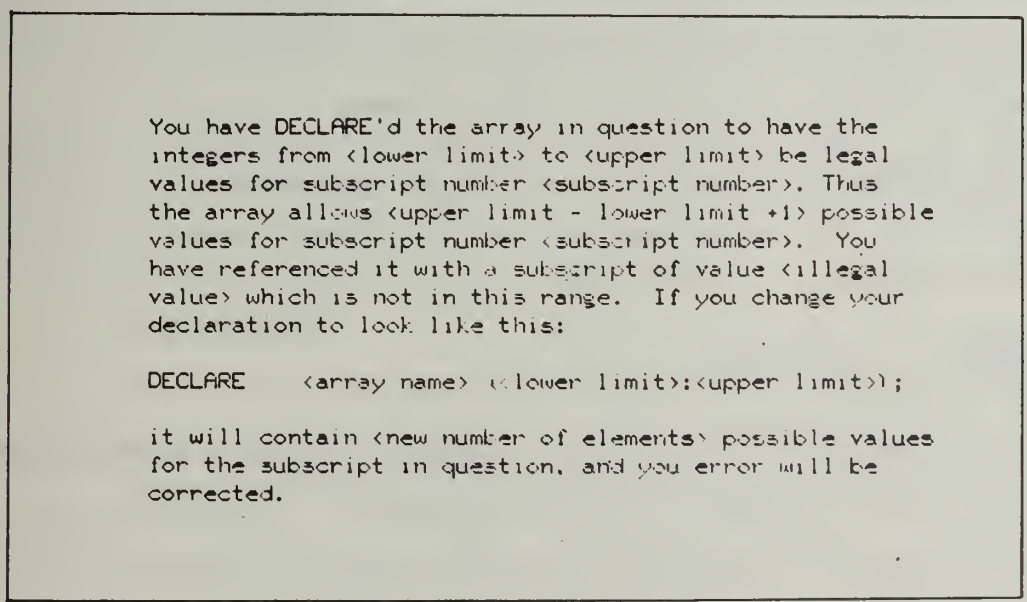


Figure 4.2 Sample Dialogue Template

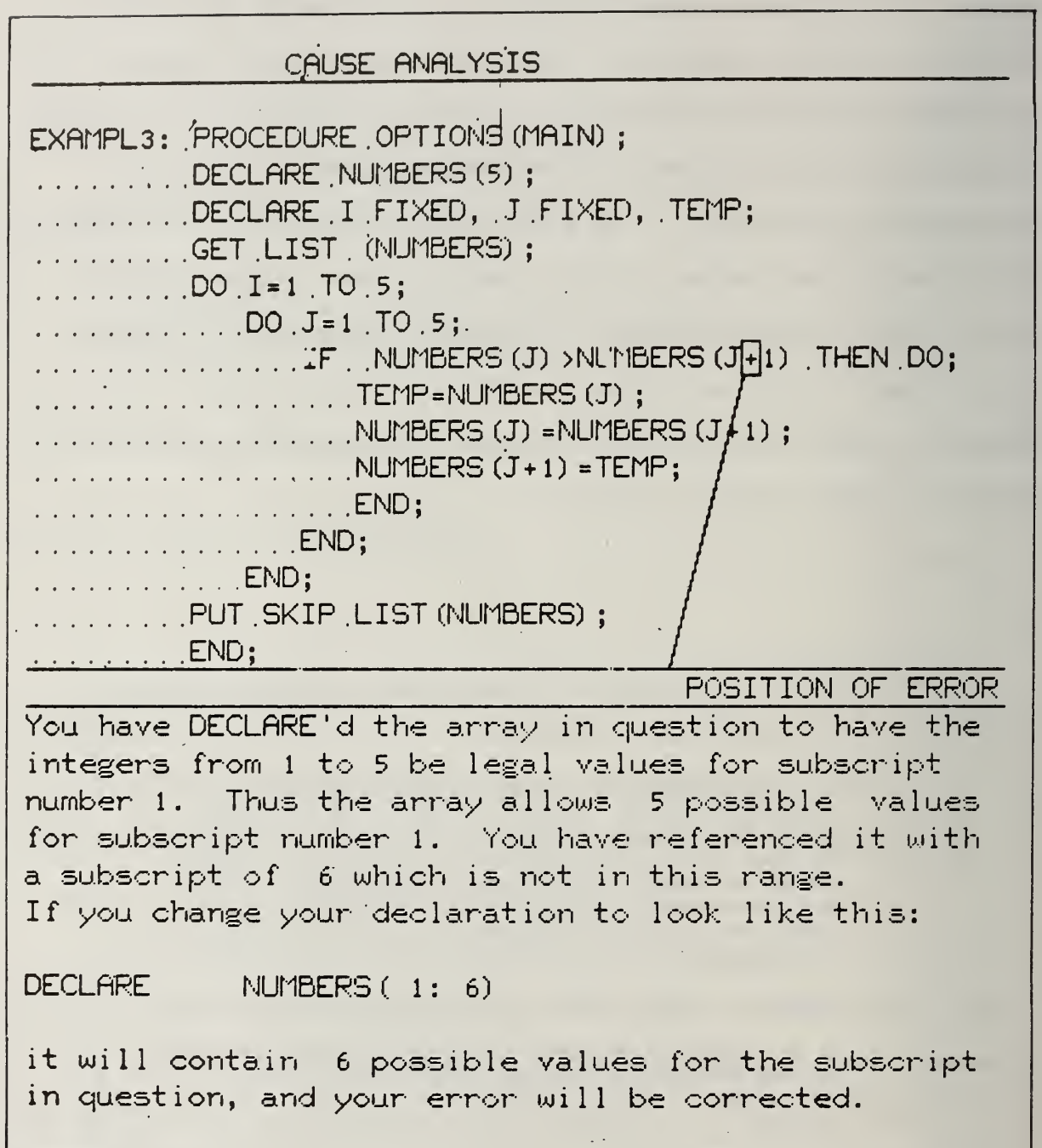


Figure 4.3 Example of Static Analysis Dialogue

Common causes of this type are termed immediate because the discussion about them is initiated immediately after the display of the list of common causes. Some common causes for an error according to the tables may or may not be relevant to the particular program. In this case, discussion is deferred until dynamic analysis if the causal condition is actually shown to be present in the program at that time. Since the discussion is deferred until dynamic analysis, common causes of this type are termed deferred.

The information about whether a particular common cause is immediate or deferred is given in the common misconception table. Also located here are the corresponding dialogue templates. The proper common misconception table entry is located by an index equal to the number of the common cause. This table holds the only language dependent information in the execution-time error analysis routines; it must be written anew for each language implemented on the programming system. It is this table that makes the error analysis system table driven. Some examples of common misconception table entries for the PL/I language implementation are given in Appendix B.

4.3.4 Implementation

In order to maintain up-to-date data on correlations between execution errors and misconceptions, it is necessary to constantly change this stored information. Although expensive (in terms of PLATO's restrictions on resources) to access, the only place for the databases to reside is on a dataset. This would be accessed exactly twice per execution error occurrence: once to fetch the present information and once to store the updated information. As said before, not only is it necessary to store

direct correlations between errors and their causes but it is also necessary to store information which is course-dependent and student-dependent.

Three databases are used:

Database 1 entries correlate error types with misconceptions over a period of n days. Entry DB1(m,t,d) contains the number of times that any student corrected an execution error of type t when being shown the discussion for the misconception m, d days ago. Thus $\sum_{d=0}^{n} DB1(m,t,d)$ is the number of times that misconception m caused error type t during the past n days and is thus some indication of the correlation between those two situations within a course of students.

Database 2 entries correlate misconceptions with individual students during an entire semester. Entry DB2(m,s) contains the number of times that student s corrected an execution error after being shown the discussion for misconception m. Thus database 2 entries are some indication of how often a particular student makes specific mistakes or how often a student misunderstands a particular concept.

Database 3 entries correlate misconceptions with actual execution errors. This database is static and does not change during a semester. In particular, DB3(k,e) for k = 1 to p is a set of at most p common causes presented automatically to all students having error e occur in their program. p would be some pre-determined limit on the maximum number of possible common causes to be associated with an execution error in database 3.

The following illustrates how these tables are used.

Assume that student s has just received an execution error e which is of type t. We are trying to find a subset of the misconceptions for a

language which correlate strongly with that error. $\sum_{d=0}^{n} DB1(m,t,d)$ tells us how often each misconception m has caused that error. Each of these values is perturbed slightly by the information in database 2 in order to hand tailor the list to the student. Thus the set,

$$C = \{m \mid \sum_{d=1}^{n} DB1(m,t,d) + perturb * DB2(m,s) \text{ is large}\}$$

is the set of all misconceptions which have a high probability of causing the particular execution error of type t. Only experimentation with an actual student community can fix the value of the constant "perturb" and the definition of "is large."

After the above set is displayed in decreasing order of value (showing the most probable at the top of the list and the least most probable at the bottom), then database 3 is examined and all

$$\{m \mid m \in DB3(k,e) \text{ for some } k \geq 1\}$$

which are not already contained in C would be appended to the end of the displayed list C. This final appendage insures that misconceptions such as "Incorrect Declaration of an Array" are included in the list for particular execution errors like "subscript out of range."

Whenever a student exits the execution-time error analysis routines, it is assumed that the discussion presently being displayed to the student has influenced the student to do so. Thus, if the student is looking at a discussion concerning a misconception m and he requests to leave error analysis, it is assumed that misconception m caused his error and the following steps are taken:

$$DB1(m,t,0) \leftarrow DB1(m,t,0) + 1$$

and

$$DB2(m,s) \leftarrow DB2(m,s) + 1.$$

In addition, as each new day starts, it is also necessary to:

$$DB1(m,t,k) \leftarrow DB1(m,t,k-1) \text{ for } \forall\, m,\ t,\ k \geq 1$$

and $$DB1(m,t,0) \leftarrow 0 \text{ for } \forall\, m,\ t.$$

This concludes the discussion of the use of the databases in the implementation which has been developed.

4.4 Dynamic Analysis

4.4.1 Introduction

Dynamic Analysis' main function is to mimic the human consultant or more often the devoted programmer/debugger when he really "gets into" debugging. This is when the debugger examines his program in minute detail, possibly doing such things as executing segments of code by hand or checking in the text book whether certain language features actually work the way that was thought. This is a very time-consuming tiresome task and it would thus be convenient to automate if possible.

Before further discussion, it will be necessary to formally define some new terms: A Bad-Valued-Variable (bvv) is a variable whose value, if altered, might prevent the occurrence of the execution error at a future time. That is not to say that its alteration will prevent it; it simply means that the alteration of any variable which is not a bvv could not possibly affect the particular error being analyzed. A Bad-Valued-Variable list contains the set of all possible bvv's at any one time. Reverse Execution of a statement is the process of returning the execution environment of the program to the exact state that it was in before that statement was executed. A Common Misconception is any facet of a higher-level language which is often misused by a beginning programmer. A full description will be given later in this chapter.

The only way to locate the source of an error which occurred as a result of a variable taking on a particular incorrect value is to reverse execute the program, searching the "history" of that variable. The process of reverse execution is already present in the execution supervisor (see section 3.3) and thus creates no additional overhead. The two problems that remain are (1) how to recognize the source of an error and (2) when to terminate reverse execution.

Recognition of the source (or cause) of an error can sometimes be done by the system. For example, in

```
X = 0;
Y = 2/X;
```

the system can easily detect that the statement X = 0; is the source of the "division by zero" execution error. Sometimes the programmer can recognize the cause of the error, as in cases where the logic of the program is concerned. The question of when to terminate is also a dual responsibility. The system should terminate when it finds the only possible cause of the error as in the case described above. Similarly, the student should terminate the process when he understands what he must do to correct his program in order to remove the execution error.

The following sections describe the process by which dynamic analysis searches for the causes of errors, the process of reverse execution, and a method of implementing the necessary tables for the process.

4.4.2 The Main Algorithm

The main dynamic analysis algorithm repeatedly reverse executes, looking for seats of possible causes for the execution error. When a

possible cause is located, an appropriate discussion about that condition is initiated. Notice that these discussions could correspond to any of the deferred-type common causes listed during the static analysis or to any other non-listed misconception which may be contained in the common misconception table.

This algorithm is invoked only when an error has occurred because a variable has taken on an illegal value. This error type was described more fully in section 4.3.2. One should also remember that almost all errors caused by an anomaly in the dynamic data structures of the program usually has its cause somewhere other than in the same position as the execution error. For errors whose cause is located in the same statement as the execution error, the common misconception table is immediately accessed during static analysis via an "immediate" common cause, and the cause is located before the following algorithm can be initiated.

Algorithm V (dynamic analysis algorithm initiated by a bad Value): We are at the occurrence of an execution error caused by a variable, v, having a bad value for its context.

Step 1 (Initialization): Push v onto the bvv list.

Step 2 (Back-up): Reverse execute one statement.

Step 3: If this statement does not modify any bvv, go to step 2; otherwise delete the bvv from bvv list.

Step 4: If this statement does not contain a (another) common misconception go to step 7.

Step 5 (Help): Enter informative dialogue with student.

Step 6 (Help sufficient?): If the student now understands the cause of his error, let him re-edit and try again; otherwise, go to step 4.

Step 7 (What does student want next?): If student wants the services of the flow exhibition, invoke it.

Step 8 (Still anomalous?): Display list of variables used in the statement. Enter all of those variables which the student is not absolutely positive have correct values into the bvv list.

Step 9 (Check): If length of bvv list is zero, tell the student that this expression being analyzed is incorrect and stop algorithm. Otherwise, go to step 2.

The following section describe how reverse execution works, how dynamic analysis figures out if a statement contains a common misconception, and what the informative dialogue with the student is. That is, the processes involved in steps 2, 4, and 5 are explained in more detail.

4.4.3 Reverse Execution

The definition of reverse execution has already been given in section 4.4.1. There are three possible types of changes to the execution environment that can be induced by the execution of a statement:

1. Change in the value of a variable
2. Change in the flow of control
3. Change in the block structuring or nesting information.

Each of these changes must be recorded in a history stack during forward execution so that they can be undone at reverse execution. The reader may think that it is necessary to store with each pushed entry a flag telling which of the three types of change is represented; however, this is not true if the stack is properly managed. That is, if it is pushed properly

during forward execution and popped correspondingly during reverse execution, the top of the stack will always contain the information needed to restore the program state in the event that the previous statement must be reverse executed. For example, if the statement were an assignment statement, then we know that the top element on the stack must be the old value of the variable being assigned.

This algorithm works fine in all cases except where a change in flow of control is concerned. It is necessary to know which entry nearest the top of stack is the last flow of control entry. This is necessary because the reverse execution algorithm, to be presented below, assumes that the next statement to be reversed is always the physically previous statement unless the last flow of control entry indicates differently.

There are three ways of knowing which of the latest entries on the stack are indicative of a change in flow of control. The first method is to save with each entry on the stack a flag indicating its type. This can immediately be eliminated because of spatial overhead.

The second is to maintain two stack pointers. One points to the usual top of stack; the other points somewhere within the stack. In fact, it would point to the latest entry indicating a change in flow of control. However, let's say that at some time we find that to reverse execute one statement we must first change our position in the intermediate text. That is, by looking at the entry in the stack corresponding to the "change in flow of control" stack pointer, we discover that the last jump (in the forward direction) was to the present position. Presumably encoded within that entry would also be the source address where the jump originated and where we want to resume reverse execution. But it is necessary to

store at every flow of control entry three items: the source address, the destination address, and a link to the position in the stack of the next entry of the same type. Thus, this method also has too large an overhead.

The third and cheapest method is to maintain just one word of overhead for the entire stack. This word, called LASTJMP, actually contains the latest source and destination addresses of the last jump in flow of control. The actual mechanism of its operation is described within the following algorithms that describe reverse execution in detail.

The data structures necessary for reverse execution are one circular stack named HISTORY which is indexed by a variable HSTCNT and one word named LASTJMP which is composed of two half-words called LASTJMP.SRC and LASTJMP.DST. HISTORY (HSTCNT) is the top of stack and HISTORY(1) is the bottom or base of the stack. Assume HSTCNT is initially set to 0.

Algorithm F (Forward execution's effects on HISTORY):

Step 1 (Change in value of variable): If this statement modifies a variable v, then HSTCNT ← HSTCNT + 1 and HISTORY (HSTCNT) ← old VALUE(v).

Step 2 (Change in nesting information): If this statement modifies the state of the program's nesting information, then push onto HISTORY stack all information needed to undo that change in a similar way as in step 1.

Step 3 (Change in flow of control): If this statement causes the next statement to be executed to be other than the physically next statement then,

HSTCNT ← HSTCNT + 1,

HISTORY (HSTCNT) ← LASTJMP,

LASTJMP.SRC ← present position in intermediate text, and

LASTJMP.DST ← position in intermediate text where the next statement to be executed resides.

Algorithm R (Reverse execution's effects on HISTORY):

Step 1 (Is the next statement to be reversed the physically previous statement?): If present position in text ≠ LASTJMP.DST then go to step 2. Otherwise present position in text ← LASTJMP.SRC, LASTJMP ← HISTORY (HSTCNT), and HSTCNT ← HSTCNT - 1.

Step 2 (Change in value of variable?): If the statement to be reversed modifies a variable v, then VALUE(v) ← HISTORY (HSTCNT) and HSTCNT ← HSTCNT - 1.

Step 3 (Change in nesting information?): If the statement to be reversed changes nesting information, restore the program states appropriately using the information on the top of HISTORY and reduce HSTCNT appropriately.

In the above algorithms, the description of the changing of nesting information was deliberately vague. The reason for this is that necessary alterations differ drastically from language to language and from statement to statement. It must simply be remembered that for a particular implementation of the above algorithms, push all information needed to restore the state of the program, and during reversal pop the same amount.

4.4.4 Searching for Common Misconceptions

Once a statement has been located which has affected a bvv, that statement must be carefully analyzed for it might be the seat of the cause of the execution error. For example, in the case of an assignment statement, it is necessary to scan the right side of the assignment to see if any conditions are present to make the error analysis routines suspicious.

So far it has been pointed out (in section 4.3.3) that the common misconception tables can be referenced during static analysis by the number of that common cause. Another method of referencing some of the common misconceptions is by an (operator, value) pair. This pair corresponds to an operator in an expression which during expression evaluation resulted in the particular value. In the most general case this access should be made using an n-tuple:

(operator type, value, relation,
operand value$_1$, condition$_1$,
operand value$_1$, condition$_2$,
.
.
.
)

where the condition$_i$ defines the necessary condition which must exist between each data entry and the indicated value and the relation defines the necessary relationship between the value and the actual value returned by the operator. In the specific case of just (operator, value), it is assumed that the relation is "equality" and that the values of the operands are unimportant. This specific case takes care of most error situations.

While an expression is being scanned, each operator and the value returned are looked up in the table. If the pair is present, the discussion corresponding to it is initiated; otherwise the scan continues. The scan of the

expressions is done by a pre-order traversal. This creates a top-down analysis of the expression so that if there is more than one common misconception in a given expression, the "outer-most" one is discussed first. Then, if the student needs more help, the inner ones are discussed to give the student more details about the situation.

The importance of this top-down approach can be seen with a simple general example. Let's say that we have

$$A = X(B, C, Y(D)),$$

where X and Y are built-in functions, which in this case returned special-case answers. Built-in function operators and their special-case results should always be included in the common misconception table for each language. Let's say in this case they have been included. If the problem with Y were discussed first, the student would probably respond with something like "so what?" because he does not see its relevance to the bad value assigned to A. However, if the problem with X were discussed first, then the student would at least see the reason that A received its illegal value. Now the student understands the reason for the discussion at this step and upon asking for more assistance would find out why X returned the value that it did; in fact, X returned its value because Y returned its special-case value. Thus, when the student asks for more help he gets it; he finds out through the discussion about Y why X's value was so strange.

The actual algorithm for the pre-order traversal is as follows:

Algorithm P (Parse the expression):

Step 1: Parse the expression into a tree. At each node have fields indicating the operator type, resulting value, whether it's a constant, and pointers to each of its operands.

Step 2: Do a pre-order traversal of the tree. At each operator (internal node), see if the operator type and resulting value are located in the common misconception table. If not, continue the pre-order traversal. If so, do step 3 and then continue the pre-order traversal.

Step 3: Initiate the appropriate discussion from the common misconception table. If the student now sees his mistake, record this fact in the data bases, and allow him to go back and re-edit his program. If the student understands this situation and does not believe it to be his mistake, do not modify the databases.

Note that there may be more than one (operator, value) pair corresponding to any one common misconception. In addition, more than one misconception may have the same pair.

Note also that the actual implementation of reverse execution must be language dependent; it is thus part of the execution supervisor and not part of the error analysis system. Because the goal is to have the error analysis language independent, it is necessary to establish some common communication between the execution supervisors' reverse execution and the error analysis routines. Because no parameters are required to be passed from error analysis to reverse execution a simple call is needed to reverse execute one statement. However, parameters passed from reverse execution to error analysis must contain the following information:

1. How many variables were given new values, if any.

2. Which variables were given new values.
3. Pointers to the expression trees which were assigned to the variables.
4. Whether reverse execution is impossible.

Notice from Algorithm V that in cases where statements simply alter flow of control or modify nesting information, dynamic analysis is unconcerned and would simply initiate another reverse execution step. At first glance, the situation described by item 4 above could conceivably arise from either of two conditions: the beginning of the program could have been reached, or it was necessary to throw away some of the HISTORY stack during forward execution due to a lack of space.

Unless the HISTORY stack overflowed (i.e., the circular stack overwrote itself), value analysis must terminate before reverse execution becomes impossible. The reason for this conclusion is simple. Implicit in the maintenance of the bvv list as described in algorithm V is the fact that although the modification of the value of any one bvv may or may not remove the execution error, it is true that there does exist some bvv in the bvv list which, if modified, will prevent the execution error. Assuming that PL/I-like INIT and FORTRAN-like DATA statements are not implemented, when the program is reversed to its beginning, the bvv list must be empty. This means that there does not exist any variable that can be modified to prevent the error; this is a contradiction because we know one exists. It exists because the dynamic analysis procedure was initiated when a subexpression had an illegal value. Thus, the bvv list contained at least one variable at the start. The bvv list length is always being checked in step 9 of algorithm V and it is not allowed to go to zero. If the only variable in the bvv list is assigned a constant, or the original subexpression was a constant, then the constant must be wrong and the

student is told so. If the only variable in the bvv list is assigned a variable expression and the student insists that all variables in that expression have correct values, then the student has made one of the following mistakes:

1. The expression we are presently examining is not the expression that the student wants to assign to the variable. That is, we have found a logic error!
2. The student does not understand how his program is supposed to be working and thus has answered falsely to the requests by the analysis program to tell which variables have absolutely correct values.

Thus dynamic analysis cannot proceed back to the beginning of the program without the student making an error in his statements to the analysis system. If the student answers all questions properly, then dynamic analysis will eventually either conclude that "THIS CONSTANT IS INCORRECT", or that "THIS EXPRESSION IS INCORRECT FOR THE LOGIC OF THE PROGRAM", or that "THE VALUE INPUT HERE IS INCORRECT."

4.4.5 Implementation

Implementation of dynamic analysis for the author was an exercise in dislike for the PLATO system's resource limitations. The problems that arose here are detailed in section 4.6 of this thesis. At this time however, let it be stated that there is no inherent reason why dynamic analysis should be difficult to implement.

In examining algorithm V it is obvious that most steps are simple to implement. Standard algorithms for searching lists and pre-order traversing of trees can be coded with little problem. The main problems

arise in the necessity to communicate with the execution supervisor at two stages during the algorithm. To reverse execute, it is necessary to call the reverse execution procedure of the execution supervisor. This reverse execution supervisor must return the appropriate parameters discussed in section 4.4.3. To create a parse tree for relevant expressions, it is necessary to activate the expression analyzer of the execution supervisor. Both of these calls to the execution supervisor are necessary because the tasks to be performed are dependent on the language being executed and analyzed. The actual dialogues resulting from dynamic analysis on the PLATO implementation can be seen in Appendix C.

Assuming that the target system is not PLATO, implementation of dynamic analysis should be straightforward. It is simply a coding of each of the algorithms described in the previous sections along with the appropriate calls to the execution supervisor. Ideally, it would be appropriate for dynamic analysis as well as all of the other error analysis routines to reside in memory along with the execution package. This would eliminate the need for the operating system to do an unneccessary amount of swapping.

4.5 Flow Exhibition

The main purpose of flow exhibition during error analysis is to provide the student with additional information during value analysis. Flow exhibition is actually an extension of the statement trace facility during execution. It is invoked by the student during value analysis whenever he wants an answer to a question of the form: "How did we get from here to there?" When an execution error occurs, the student knows where it is located and the error analysis routines know where it is located, too. Similarly, when dynamic analysis locates a position in the

program which should be discussed with the student, that position is referenced in such a way that the student also knows what is being discussed. Thus, at all times during value analysis it is clear what physical location is being discussed.

A problem arises, however, because an executing program is a dynamic process. Although the student knows, for example, that point x is being discussed, he might not know which activation of that point is being discussed. For example, if an error occurs somewhere beyond the domain of an iterative loop and some statement within that loop is being discussed, the student does not know whether the discussion of the statement pertains to the first, second, third, etc., iteration of that loop. This, of course, is a simple example of the general case where the student wants to know how program execution proceeded from the point of discussion to the position of error. The error analysis program should be capable of supplying such information.

However, flow exhibition does more than just supply a trace. While flow exhibition is showing the flow of control, the student should be able to stop at any time and ask any of a set of questions about that flow of control. This set of questions would depend on the particular language being executed but would typically include in questions such as:

1. Why was the THEN(ELSE) clause executed instead of the ELSE(THEN) clause?
2. Why was(n't) the DO loop executed again?

In general, they are questions about why particular branches did or did not take place.

If, as stated above, flow exhibition is language dependent, how can it be part of the error analysis package which is language independent? The answer is simple: it is actually part of the execution supervisor. It should simply be an extension of the features already available during statement trace mode. It is being discussed here only because it is a necessary feature to be available to the student during error analysis. From the student's point of view, flow exhibition is part of the error analysis. From the system's point of view, it is not. Although the software for flow exhibition resides in the execution supervisor it provides two valuable services: it expands the information that the statement trace supplies during normal execution AND it gives the student more helpful information during dynamic analysis.

The choice of questions to be available as well as when they are to be available is a decision to be made during implementation. Since these are language dependent, each execution supervisor would have its own flow exhibition. Associated with each statement which alters the flow of control must be a corresponding set of questions about that statement. Thus associated with each section of code within the execution supervisor which executes a particular type of statement must be another section of code which can respond to questions about why the statement did or did not alter the flow of control.

It is easiest to implement flow exhibition by the use of a single flag. If the flag is clear, the execution supervisor simply executes the statement normally. If the flag is set, the same section of the execution supervisor executes the statement, but it also answers all the appropriate questions as it is executing. This eliminates the necessity of storing intermediate results to be used later by the flow exhibiter.

The main drawback of this is that the student must now ask the questions about how a branch is going to be executed before it happens. This is not considered a major drawback because the student upon witnessing a questionable alteration of flow of control, either during trace or during value analysis flow exhibition, may stop, reverse execute one statement, ask the appropriate questions and then continue the flow. Another trait of this implementation is that there does not exist a routine whose task is to answer the appropriate questions; instead the question answerers are spread among the sections of the execution supervisor which interpret the particular statements. The actual module in the execution-time error analysis routines called flow exhibition is simply a list of calls to activate the appropriate trace facilities that are present in the execution supervisor.

4.6 Implementation Problems

The implementation of this analysis system on PLATO challenged every one of PLATO's resource allocation limitations. The large number of special-interest groups involved makes it understandably impractical for PLATO management to modify the system for individual users. Here is a list of some of these limitations. Following the list is a description of how the problems were (or were not) overcome.

1. Program binary must not be greater than 8000 words.
2. Total addressable memory must not be greater than 1500 words.
3. Total amount of additional storage allocated to a user must not be greater than 1500 words.
4. Only 1000 words will be allocated per terminal per site.
5. Data sets may be used for unlimited back-up storage but require a disk access (see 7 below).

6. Jumps from one program to another could overcome problem 1 but count as a disk access (see 7 below).
7. Disk accesses will be limited to only a few every 1/2 hour per user.
8. CPU usage limited to 2-3msec/sec.

Because PLATO management remained unwilling to modify the 8000 - word program size limit, it was necessary to break the execution-time error analysis routines into a program separate from the execution supervisor because the execution supervisor was about 8000 words in length for PL/I. However, there is a necessity for the dynamic analysis routines to communicate with the execution supervisor's expression analyzer (to receive parse trees for expressions) and its reverse execution routines (to reverse execute one statement at a time). Although the band width of transferred arguments is narrow, there is a necessity for many transfers of control between dynamic analysis and the execution supervisors. This seems to be little to ask of a supporting operating system and yet PLATO makes it practically impossible for flow of control to pass between two large programs.

The only way to perform this transfer of control is by a JUMPOUT command which (1) counts as a disk access, and (2) may require a compilation (condensation) of the target program. The disk access rate as seen in 7 above is very restrictive. In fact, dynamic analysis and the execution supervisor may need to JUMPOUT to each other hundreds of times during a thirty minute debugging session! Worse than this is the possible compilation of the destination program. It is unlikely that this would be necessary but if it were, it might take as much as ten minutes for a single JUMPOUT during midafternoon peak-user hours! We obviously have a major problem here. The analysis routines were originally designed to be USE'd by the

execution supervisor. USE is TUTOR's equivalent of a macro call which is expanded at compile time. Unfortunately, this brings the program size over 8000 words. The solution settled on was to USE the reverse execution routines of the execution supervisor in the error analysis program. This successfully cuts down the number of necessary jumpouts to a reasonable number. Now there is a conflict, however, with one of the error analysis' design goals: to make it language independent. In its present mode of operation, a seperate program containing the error analysis routines for each higher-level language implemented is needed. Their source code would be identical except for the USE commands which would USE the reverse execution modules from the appropriate execution supervisor. There are only two other solutions: to have PLATO alter its lesson size limitations, or to explore alternate installations.

The problem of PLATO's addressable data area being limited to 1500 words was not a major problem due to complex overlaying procedures that were used during compilation, execution, and analysis. All three processes use different overlay mechanisms and transfers of control from one to another required careful attention to the memory space. This overlaying has been successful due only to the convenient TUTOR commands which make swapping between addressable memory space and back-up storage user controlled.

The fact that PLATO will allocate only 1000 words per terminal per site places a severe limitation on the entire programming system. The compiler itself uses about 8000 words, the execution supervisor for PL/1 also uses about 8000 words, the back-up storage associated with the set of lessons, called COMMON is about 1300 words, the student filing system is about 3000 words, the compile-time error analysis routines are 4000 words and the

execution-time error analysis routines are about 4000 words. In addition, the system requires an additional 1500 words of storage allocated to each terminal. Thus we have an overhead approximately 25,000 words as well as 1500 words per terminal. In order to comply with PLATO's 1000 words per terminal per site restriction, it would be necessary to reserve a site with (25 + 1.5n) terminals in order to permit n students to run on the system at one time! In the spring of 1975, 600 students will be using the compiler system for the first time. The only two possible solutions are to convince PLATO to alter their allocation scheme, or else reserve something like 70 terminals for 30 students. What a waste of terminals!

Because of storage limitations, it was not easy to find room to store the relatively large data bases necessary for static analysis. These databases should contain the following numbers of words:

DB1 = number of common causes (≈20)
x number of error classes (≈10)
x number of days recorded (≈5)
x number of bits per entry (≈10)
= 10,000 bits = 166 words

DB2 = number of common causes (≈20)
x number of students (≈600)
x number of bits per entry (≈10)
= 120,000 bits = 2000 words

DB3 = number of execution errors (≈40)
x maximum number of common causes
allowed per execution error (≈5)
x number of bits per entry (≈5)
= 1000 bits = 17 words

Since only 2 accesses are necessary per execution error (once to access the data and once to replace it with the updated copy), the most reasonable place to store these 2183 words is on a dataset. For this particular experimental implementation, the total database size was cut down to just 122 words and was stored along with other system tables in common storage. This considerable reduction in size was made possible by making the following temporary limitations:

number of common causes $\leq$ 20

number of execution errors $\leq$ 60

number of error classes $\leq$ 20

number of students $\leq$ 26.

5. RESEARCH CONCLUSIONS AND FURTHER RESEARCH

5.1 Error Analysis Research (Restrictions and Improvements)

As section 4.6 pointed out quite clearly, the PLATO system is not ideal for compiler construction or error analysis. In fact, the system is only barely adequate. Hopefully, further research in error analysis will be tested on systems which are more suitably designed for this type of work.

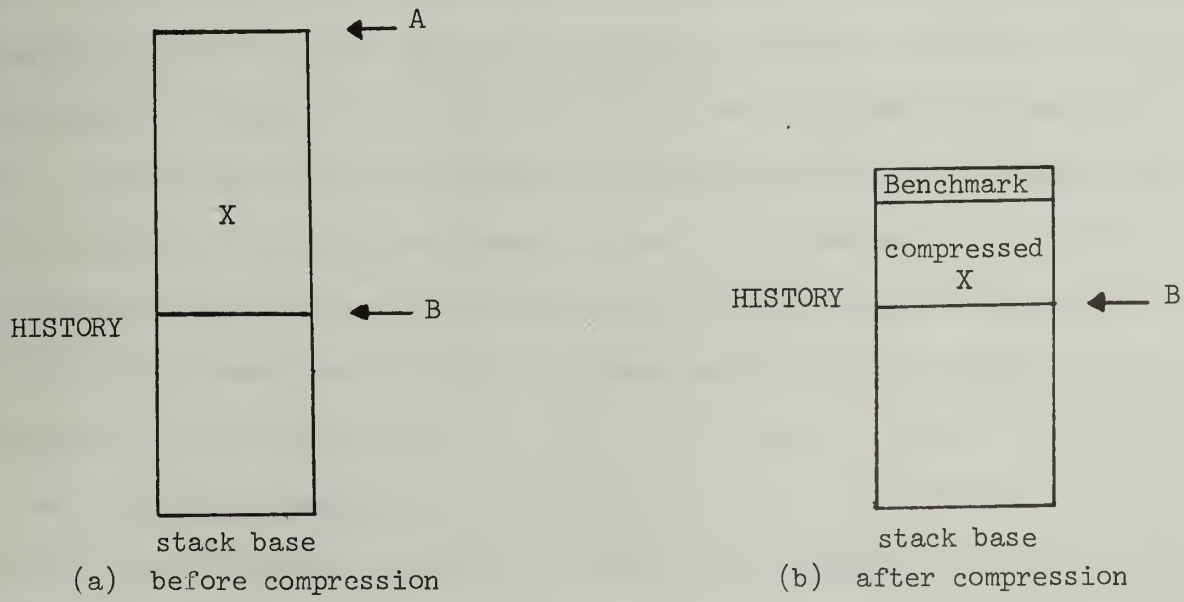
The present implementation was designed as an experimental system. As Appendix C indicates, the implementation as it stands is quite effective in locating bugs in simple programs. Data area limitations have limited the capabilities of the system significantly however: the circular buffer used to hold reverse execution's HISTORY must be restricted in size. In the PL/1 execution supervisor and error analysis package there exist only 180 words in which to store the student's activation records and the HISTORY stack. Approximately one word is pushed onto the HISTORY stack for each statement executed. Therefore, we arrive at the following restriction:

Number of words allocated to student variables + number of statements which can be reverse executed during dynamic analysis $\leq$ 180.

This restricts the set of programs which we may analyze completely. In fact, we can pinpoint a cause of an error only if it is located within a particular distance from the error. This situation must certainly be corrected before introducing the system to a production environment where

systems programmers, for example, can debug complex pieces of software. There are two alternatives to the present situation where only the last n entries of the HISTORY stack are saved:

1. Spool the HISTORY stack to a disk data set.
2. Compress the HISTORY stack somehow whenever the space overflows. The basic idea behind this would be to remove some set of entries from the HISTORY stack and replace it with a benchmark to indicate that a compression had taken place. Along with that benchmark would be information about the entries compressed so as to enable reverse execution to proceed. For example, say at some point A, a compression is necessary (Figure 5.1a). Then replace all entries from point B to point A with a compressed version of those entries as in Figure 5.1b. During reverse execution if a benchmark is discovered, all statements until point B are reverse executed in one batch by using the information stored in the compressed X. The program is then forward executed until the stack fills up again. At this point we have returned the HISTORY stack to the position



(a) before compression (b) after compression

Figure 5.1 Compression of HISTORY stack.

in Figure 5.1a. Now reverse execution may proceed normally. The information stored in the compressed area of the stack must include one entry for each variable which is changed any number of times since the time of execution at point B (this entry must include the variable's identification as well as its value at point B), the position in the intermediate text corresponding to point B, and a resolution of all nesting changes that have occurred between points B and A. This compression method is similar to that described by Zelkowitz [24].

At present, the databases described in section 4.3 are updated every time a student exits the error analysis routines. The problems with this are that the student may either be making the wrong correction for his error or may be responding to earlier advice from the system. The former problem can be alleviated by recording for each student the latest execution error and correction that he made to his program. If the identical error re-occurs, it can be assumed that the previous correction was not proper and the data bases can be appropriately decremented. The latter problem should occur only occassionally. These databases are used only for static analysis and static analysis relies heavily on probability. That is, if many students correct an execution error in response to a discussion about a particular common misconception, then it is assumed that common misconception is a good candidate as a common cause for that error. Therefore, occassional spurious data would be hidden by a large amount of significant data.

Another place where the implemented algorithms can be improved is in the actual design of the common misconception tables. Due to a lack

of storage space, it was decided to hard-code the common misconception tables in TUTOR. The dialogue templates contained in these tables are presently coded basically as conditional write (WRITEC) commands with embedded SHOW's to hand-tailor the discussions to the particular program. A far more general method would be to store it as a table in common storage along with all the other tables which define the language. That way all the tables would be accessed in the same general fashion. The only difference between the execution-time error analysis routines for one language and another would be the common storage that each used.

5.2 Research Conclusions

There is some difficulty in arriving at a conclusive evaluation of the performance of the execution-time error analysis system because of the lack of extensive use by students. Since the system has been designed to assist beginning programming students with debugging, objective statements can be made only after such use.

However, some preliminary comments based on observation can be made. When first investigated it appeared that static analysis, dynamic analysis and flow exhibition were all of equal importance. After careful study and actual implementation and use, it appears that dynamic analysis stands out, from the user's point of view, as the one that is the most dramatic and revolutionary in approach. Flow exhibition has taken a very subservient position as being a tool within a tool within a tool. (That is, it is activated from within the execution supervisor's trace mode by the dynamic analysis procedure.)

5.2.1 Static Analysis

It is difficult to say at this time whether static analysis

is going to be cost-effective. To be adequately tested, its databases must contain a large amount of relevant data. The data collected during testing by the system developers and evaluaters is not adequate; only actual student use can control the generation and authenticity of these databases. Until that time, only static database 3 can be evaluated properly. In actual use, this database catches various causes of execution errors immediately. In particular, it catches causes which can be discussed statically; that is, those that require no reverse execution to find. Typically, these include all types of bad declarations for variables (e.g., improper definition of subscript bounds for arrays). During observation of the few users of the system so far, it is obvious that the static analysis, as activated by database 3 only, is both meaningful and essential.

The static analysis, as activated by databases 1 and 2, has debatable value. When fully implemented, this requires 2000 words of storage. Since PLATO has such severe memory limitations, this raises serious cost-effectiveness questions. Deferment of the answers must be made pending actual use. Since its main purpose is to save computer time by avoiding the lengthy processing necessary during value analysis, its practicality can be decided only after seeing how many students' execution errors can be corrected by it alone. If only a few of the students' execution errors can be corrected by static analysis by effectively prompting them to examine their programs for certain conditions, then the process would be worthwhile. However, if all students simply ignore the list and continue immediately on to dynamic analysis, then perhaps static analysis should be deleted from future versions.

5.2.2 Dynamic Analysis

During the limited testing, dynamic analysis has been quite effective. In all test programs used, value analysis did locate the possible causes of the error whenever the static analysis did not. When dynamic analysis was being used, the logic followed was quite similar to the human debugger's logic. Dynamic analysis is thus quite useful to the beginning programmer to assist in the debugging process as well as to teach him how to properly debug. For more experienced programmers, it saves debugging time by screening the program and pointing out only possible troublespots. The analysis routine is more efficient at debugging than the beginning programmer for two reasons. First, it has remembered exactly how the program was executed. Thus it knows exactly where variables were given values. By asking simple questions of the programmer, it can easily locate spots where the programmer has misunderstood certain features of the language. Second, it knows what types of errors are common; it has this knowledge from its various tables. Thus it seems that the dynamic analysis is potentially better at debugging than a human. Actual experimentation seems to verify this fact. All students who tried the system were pleased with its obvious advantages over a traditional batch system's error detection and correction methods.

5.2.3 Flow Exhibition

As pointed out in section 4.5, flow exhibition is actually a part of the exhibition supervisor. At the time when it was realized that this was true, the PL/I execution supervisor was just 2 words short of PLATO's upper limit on program length. Although flow exhibition has been implemented to supply a trace between any two specific positions in a program, the feature of letting the student interrupt and ask specific

questions has not. The reasons for this are (1) there did not exist any room at all in the PL/I execution supervisor and still remain within PLATO's confinements, (2) flow exhibition has, after considerable thought, become less important as a tool during value analysis than as a tool during actual execution, and (3) thus, becomes of very minor importance in verifying the practicality and effectiveness of the theories and research expressed in this thesis.

5.2.4 PLATO

The PLATO terminal must be mentioned here as a very effective interactive display medium. The panel permits quick display of both textual information and arbitrailly complex diagrammatic output. Without its diverse capabilities, the error analysis routines would have suffered considerably in their ability to express what was desired. It is a shame that the computing power of the PLATO system cannot match the terminal in its ability to provide good service to the compiler project.

This is the first attempt ever to automate the analysis of execution-time errors. As such, it should be viewed as a preliminary investigation into a brand new field of computer science. It is hoped that other individuals, as well as myself, will find it useful to expand the horizons set by this work.

LIST OF REFERENCES

[1] Albert, D. and D. L. Bitzer, "Advances in Computer Based Education," Science, Vol. 167 (20 March,1970), pp. 1582-1590.

[2] Balzer, R. M., "EXDAMS-Extendable Debugging and Monitoring System," AFIPS Conference Proceedings, Vol. 34 (1969), pp. 567-580.

[3] Bernstein, W. A. and Owens, J. T., "Debugging in a Timesharing Environment," AFIPS Conference Proceedings, Vol. 33 (1968), pp. 7-14.

[4] Carr, C. S. FORTRAN II Reference Manual, Document #30.50.50, University of California, Berkeley (Feb., 1966).

[5] Conway, R. W. and Wilcox, T. R., "Design and Implementation of a Diagnostic Compiler for PL/I," CACM, Vol. 16, Number 3 (March, 1973), pp. 169-179.

[6] Dunn, T. M. and Morrissey, J. W., "Remote Computing-An Experimental System," AFIPS Conference Proceedings, Vol. 25 (1964), pp. 413-424.

[7] Evans, T. G. and Darley, D. L., "Online Debugging Techniques: A Survey," AFIPS Conference Proceedings, Vol. 28 (1966), pp. 37-50.

[8] Fabry, R. S., "MADBUG--A MAD Debugging System," in The Compatible Time-Sharing System, A Programmer's Guide, 2nd edition, MIT Press, Cambridge, Mass., 1965.

[9] Gries, D., Compiler Construction for Digital Computers, John Wiley and Sons, Inc., New York, 1971, pp. 456-458.

[10] Jacoby, K. and Layton, H., "Automation of Program Debugging," paper presented at the 16th National Conference ACM, Los Angeles, California, September 8, 1961. Abstract in CACM, Vol. 4 (1961), p. 7.

[11] Kulsrud, H. E., Helper--An Interactive Extensible Debugging System, IDA--Communications Research Division Working Paper No. 258.

[12] Lafrance, J. E., Syntax-Directed Error Recovery for Compilers, Ph.D. Thesis, Computer Science Departmental Report #459, University of Illinois, Urbana, Illinois, 1971.

[13] Levy, J. P., Automatic Correction of Syntax Errors in Programming Languages, Ph.D. Thesis, Technical Report #TR71-116, Computer Science Department, Cornell University, Ithaca, New York, December, 1971.

[14] Martin, W. and Hart, T., Time-Sharing LISP, Massachusetts Institute of Technology Memo MAC-M-153 (rev. 1964).

[15] Miller, J. C. and Maloney, C. J., "Systematic Mistake Analysis of Digital Computer Programs," CACM, Vol. 6, Number 2 (Feb., 1963), pp. 58-63.

[16] Nievergelt, J., Reingold, E. M., and Wilcox, T. R., "The Automation of Introductory Computer Science Courses," A. Gunther, et al. (editors), International Computing Symposium 1973, North-Holland Publishing Co., 1974.

[17] Schwartz, Jacob T., "An Overview of Bugs," Debugging Techniques in Large Systems, Randall Rustin (editor), Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1971.

[18] Stockham, T. G., Jr., "Some Methods of Graphical Debugging," Proc. IBM Scientific Computing Symposium on Man-Machine Communications, Thomas J. Watson Research Center, Yorktown Heights, May 3-5, 1965.

[19] Tindall, Michael H., An Interactive Table-driven Parser System, masters thesis, Department of Computer Science, University of Illinois, Urbana, Illinois, January, 1975.

[20] _____, Table-driven Compiler Error Analysis, unpublished Ph.D. Thesis Proposal, Department of Computer Science, University of Illinois, Urbana, Illinois, April, 1974.

[21] _____, Ph.D. Thesis to be published June, 1975, Department of Computer Science, University of Illinois, Urbana, Illinois.

[22] Wilcox, T. R., "The Interactive Compiler as a Consultant in the Computer Aided Instruction of Programming," Proceedings of the Seventh Annual Princeton Conference in Information Sciences and Systems, March, 1973.

[23] Wolman, B. L., "Debugging PL-1 Programs in the Multics Environment," AFIPS Conference Proceedings, Vol. 41, Part I (1972), pp. 507-514.

[24] Zelkowitz, M., Reversible Execution as a Diagnostic Tool, Ph.D. thesis, Department of Computer Science, Cornell University Technical Report, January, 1971.

APPENDICES

A. SAMPLE DATABASES

DATABASE1

---error classes---	---common causes---														
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø
2	3	1Ø	2	Ø	Ø	Ø	Ø	Ø	1	Ø	Ø	1	Ø	Ø	Ø
3	4	1	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø
4	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø
5	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø
6	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø

DATABASE_2

---student---	---common causes---														
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø
2	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø
3	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø
4	3	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	1	Ø	Ø	Ø	Ø	Ø
5	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø
6	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø
7	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø
8	Ø	1	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø
9	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø
10	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø
11	Ø	1	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø
12	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø
13	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø
14	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø
15	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø
16	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø
17	Ø	1	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	1	Ø	Ø	Ø
18	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø
19	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø
20	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø
21	3	6	2	Ø	Ø	Ø	Ø	Ø	1	Ø	Ø	Ø	Ø	Ø	Ø
22	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø
23	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø
24	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø
25	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø
26	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø

ASSIGNMENT OF ERRORS (DATABASE 3)

What error do you want to change? ⟫
(Press [BACK] to return to index;; [DATA] to see more entries.)

error number	error class	default common causes
1	2	
2	4	10
3	3	
4	1	7
5	3	5
6	3	5
7	5	6
8	2	
9	1	
10	3	4
11	3	
12	0	
13	0	
14	0	
15	0	
16	0	
17	0	
18	0	
19	0	
20	0	

NOTE: Error classes 1, 2, 3 and 4 are RESERVED for bvv<0, bvv=0, bvv>0, and uninitialized variable respectively.

--PL/I MISCONCEPTION TABLE---

	immediate?	(operator type, value)
1.Incorrect constant in program.	NO	(Ø, Ø.ØØØ)
2.Incorrect expression in program.	NO	(Ø, Ø.ØØØ)
3.Incorrect value input.	NO	(Ø, Ø.ØØØ)
4.Improper DECLAR'ation for length of CHAR	YES	(Ø, Ø.ØØØ)
5.Improper DECLARE'ation	YES	(Ø, Ø.ØØØ)
6.Badly formatted input.	YES	(Ø, Ø.ØØØ)
7.You have forgot an ABS	NO	(Ø, Ø.ØØØ)
8.Bad lower limit on Do loop.	NO	(Ø, Ø.ØØØ)
9.INDEX returning a value of Ø.	NO	(69, Ø.ØØØ)
1Ø.Failed to initialize variable.	YES	(Ø, Ø.ØØØ)
11.LENGTH returning a value of Ø.	NO	(65, Ø.ØØØ)
12.VERIFY returning a value of Ø.	NO	(68, Ø.ØØØ)
13.VERIFY returning a value of 1.	NO	(68, 1.ØØØ)
14.SUBSTR returning the null string.	NO	(66, Ø.ØØØ)
15.Unexpected truncation during integer division.	NO	(39, Ø.ØØØ)

C. SAMPLE DIALOGUES

C.1 A Simple Example of Dynamic Analysis

PL/I EXECUTION

```
EXAMPL1: .PROCEDURE .OPTIONS(MAIN) ;
 .........DECLARE STRING CHAR(10) VARYING, POS FIXED;
 .........STRING='ABCDE';
 .........POS=INDEX(STRING,'X');
 .........STRING=SUBSTR(STRING,POS) ;
 .........END;
```

Execution Error: INDEX OF SUBSTRING < 1.
Press HELP for error analysis; BACK for list of other options.

EXECUTION-TIME ERROR ANALYSIS

```
EXAMPL1: PROCEDURE OPTIONS(MAIN);
         DECLARE STRING CHAR(10) VARYING, POS FIXED;
         STRING='ABCDE';
         POS=INDEX(STRING,'X');
         STRING=SUBSTR(STRING,POS);
         END;
                                          POSITION OF ERROR
```

This statement gave POS an incorrect value of 0.

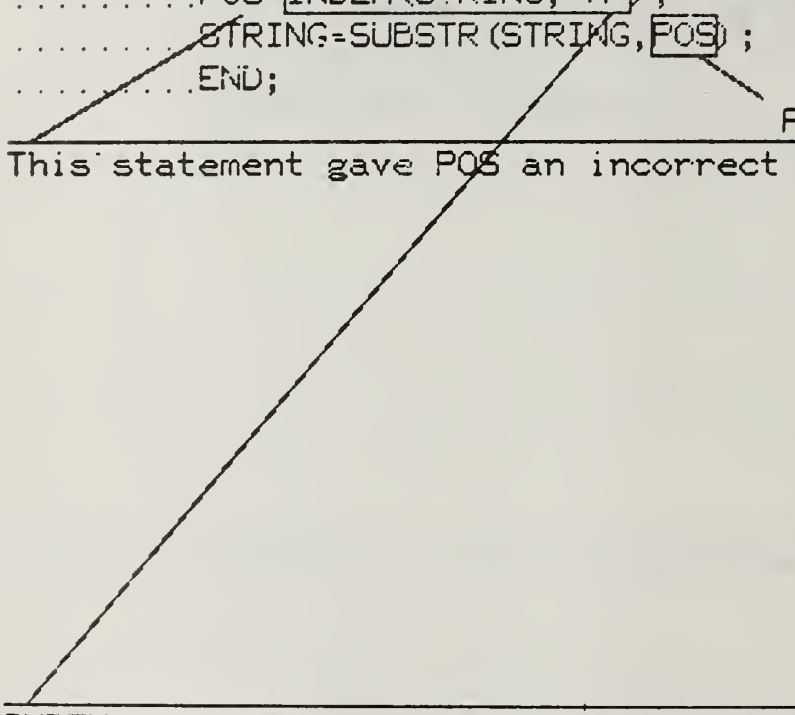
Press BACK to edit your program; HELP for more help.
Press DATA to see how execution flowed from here to error.

EXECUTION-TIME ERROR ANALYSIS

```
EXAMPL1: .PROCEDURE .OPTIONS (MAIN) ;
. . . . . .DECLARE .STRING .CHAR (1Ø)  VARYING, .POS .FIXED;
. . . . . .STRING='ABCDE';
. . . . . .POS=INDEX (STRING, 'X') ;
. . . . . .STRING=SUBSTR (STRING, POS) ;
. . . . . .END;
```

POSITION OF ERROR

This statement gave POS an incorrect value of Ø.



INDEX returned a value of zero here because the string 'X' could not be found anywhere in the string 'ABCDE'.

Press BACK to edit your program; HELP for more help.

EXECUTION-TIME ERROR ANALYSIS

```
EXAMPL1: .PROCEDURE .OPTIONS (MAIN) ;
.........DECLARE .STRING .CHAR (10) .VARYING, .POS .FIXED;
.........STRING='ABCDE';
.........POS=INDEX (STRING, 'X') ;
.........STRING=SUBSTR (STRING,POS) ;
.........END;
```

POSITION OF ERROR

This statement gave POS an incorrect value of 0.

Does a value of 'ABCDE' look reasonable for STRING?
Answer no unless you are absolutely positive!

Type y or n.

If a student answers 'yes', dialogue continues on next page.
If not, dialogue continues on page 78.

EXECUTION-TIME ERROR ANALYSIS

```
EXAMPL1: .PROCEDURE .OPTIONS (MAIN) ;
.........DECLARE .STRING .CHAR (1Ø) .VARYING, .POS .FIXED;
.........STRING='ABCDE';
.........POS=INDEX (STRING, 'X') ;
.........STRING=SUBSTR (STRING, POS) ;
.........END;
```

POSITION OF ERROR

This statement gave POS an incorrect value of Ø.

On the assumption that all of your previous responses were true, the indicated expression contains a logic error. The reason for this is that you say that all variables contained within it contain correct values, but the destination variable may not contain the resulting value.

Press BACK to edit your program; HELP for more help.

EXECUTION-TIME ERROR ANALYSIS

```
EXAMPL1: .PROCEDURE .OPTIONS (MAIN) ;
.........DECLARE .STRING .CHAR (10) .VARYING, .POS .FIXED;
.........STRING='ABCDE';
.........POS=INDEX (STRING, 'X') ;
.........STRING=SUBSTR (STRING,POS) ;
.........END;
                                        POSITION OF ERROR
```

This statement gave POS an incorrect value of 0.

On the assumption that all of your previous responses were true, the indicated expression contains a logic error. The reason for this is that you say that all variables contained within it contain correct values, but the destination variable may not contain the resulting value.

No more help available. Press BACK to edit; LAB to rerun.

EXECUTION-TIME ERROR ANALYSIS

```
EXAMPL1: .PROCEDURE .OPTIONS (MAIN) ;
.........DECLARE .STRING .CHAR (1Ø) .VARYING, .POS .FIXED;
.........STRING='ABCDE';
.........POS=INDEX (STRING, 'X') ;
.........STRING=SUBSTR (STRING,POS) ;
.........END;
                                    POSITION OF ERROR
```

This statement gave STRING an incorrect value of 'ABCDE'.

Press BACK to edit your program; HELP for more help.
Press DATA to see how execution flowed from here to error.

EXECUTION-TIME ERROR ANALYSIS

```
EXAMPL1: PROCEDURE OPTIONS(MAIN);
         DECLARE STRING CHAR(10) VARYING, POS FIXED;
         STRING='ABCDE';
         POS=INDEX(STRING,'X');
         STRING=SUBSTR(STRING,POS);
         END;
```

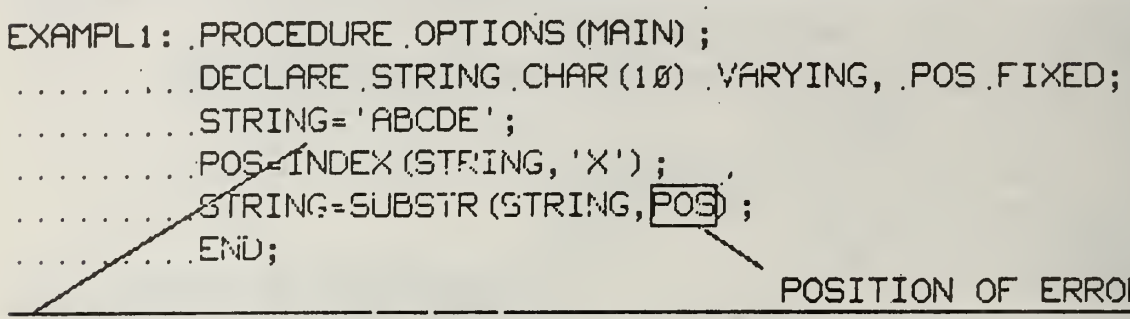
POSITION OF ERROR

This statement gave STRING an incorrect value of 'ABCDE'.

On the assuption that all your previous responses were correct, the indicated expression is an incorrect constant for the logic of your program.

Press BACK to edit your program; HELP for more help.

EXECUTION-TIME ERROR ANALYSIS



This statement gave STRING an incorrect value of 'ABCDE'.

On the assuption that all your previous responses were correct, the indicated expression is an incorrect constant for the logic of your program.

No more help available. Press BACK to edit; LAB to rerun.

C.2 Another Simple Example of Dynamic Analysis

PL/I EXECUTION

```
EXAMPL2: .PROCEDURE .OPTIONS (MAIN) ;
.........DECLARE .A .FIXED;
.........GET .LIST . (A) ;
.........A=1/A;
.........A=1/0;
.........END;
```

Execution Error: DIVISION BY ZERO.
Press HELP for error analysis; BACK for list of other options.

EXECUTION-TIME ERROR ANALYSIS

```
EXAMPL2: .PROCEDURE .OPTIONS (MAIN) ;
. . . . . .DECLARE .A .FIXED;
. . . . . .GET .LIST . (A) ;
. . . . . .A=1/A;
. . . . . .A=1/A;
. . . . . .END;
```

POSITION OF ERROR

This statement gave A an incorrect value of Ø.

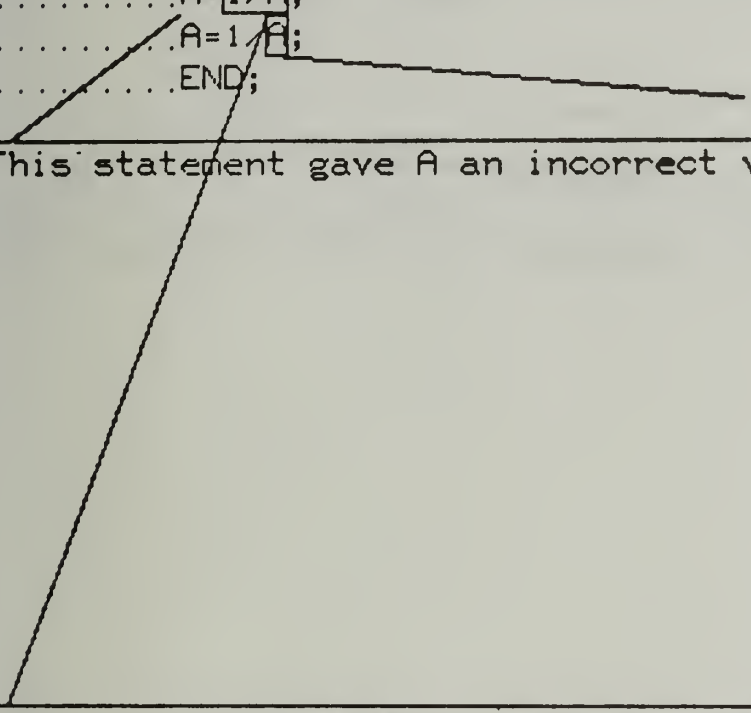
Press BACK to edit your program; HELP for more help.
Press DATA to see how execution flowed from here to error.

EXECUTION-TIME ERROR ANALYSIS

```
EXAMPL2: .PROCEDURE .OPTIONS (MAIN) ;
. . . . . .DECLARE .A .FIXED;
. . . . . .GET .LIST . (A) ;
. . . . . .A=1/A;
. . . . . .A=1/A;
. . . . . .END;
```

POSITION OF ERROR

This statement gave A an incorrect value of Ø.



This division operation performed here was integer division. In this case, the resulting value was zero because the numerator was less than the denominator.

Press BACK to edit your program; HELP for more help.

EXECUTION-TIME ERROR ANALYSIS

```
EXAMPL2: .PROCEDURE .OPTIONS (MAIN) ;
. . . . . .DECLARE .A .FIXED;
. . . . . .GET .LIST . (A) ;
. . . . . .A=1 /A;
. . . . . .A=1 /A;
. . . . . .END;
                                    POSITION OF ERROR
```

This statement gave A an incorrect value of Ø.

Does a value of 8 look reasonable for A?
Answer no unless you are absolutely positive!

Type y or n.

If a student answers 'yes', dialogue continues on next page.
If not, dialogue continues on page 87.

EXECUTION-TIME ERROR ANALYSIS

```
EXAMPL2: .PROCEDURE .OPTIONS (MAIN) ;
.........DECLARE .A .FIXED;
.........GET .LIST . (A) ;
.........A=[1 /A];
.........A=1 /[A];
.........END;
```

POSITION OF ERROR

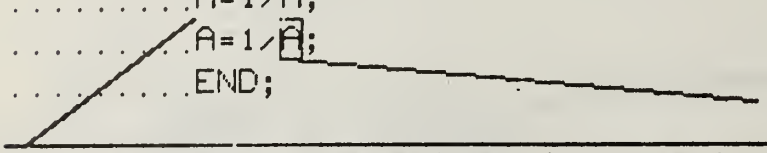
This statement gave A an incorrect value of Ø.

On the assumption that all of your previous responses were true, the indicated expression contains a logic error. The reason for this is that you say that all variables contained within it contain correct values, but the destination variable may not contain the resulting value.

Press BACK to edit your program; HELP for more help.

EXECUTION-TIME ERROR ANALYSIS

```
EXAMPL2: .PROCEDURE .OPTIONS (MAIN) ; .
. . . . . .DECLARE .A .FIXED;
. . . . . .GET .LIST . (A) ;
. . . . . .A=1/A;
. . . . . .A=1/A;
. . . . . .END;
```



POSITION OF ERROR

This statement gave A an incorrect value of Ø.

On the assumption that all of your previous responses were true, the indicated expression contains a logic error. The reason for this is that you say that all variables contained within it contain correct values, but the destination variable may not contain the resulting value.

No more help available. Press BACK to edit; LAB to rerun.

EXECUTION-TIME ERROR ANALYSIS

```
EXAMPL2: .PROCEDURE .OPTIONS (MAIN) ;
. . . . . .DECLARE .A .FIXED;
. . . . . .GET .LIST . (A) ;
. . . . . .A=1/A;
. . . . . .A=1/A;
. . . . . .END;
```

POSITION OF ERROR

This statement gave A an incorrect value .

Press BACK to edit your program; HELP for more help.
Press DATA to see how execution flowed from here to error.

EXECUTION-TIME ERROR ANALYSIS

```
EXAMPL2: .PROCEDURE .OPTIONS (MAIN) ;
. . . . . .DECLARE .A .FIXED;
. . . . . .GET .LIST . (A) ;
. . . . . .A=1/A;
. . . . . .A=1/A;
. . . . . .END;
```

POSITION OF ERROR

This statement gave A an incorrect value .

The value input here for the variable mentioned above is incorrect. Re-execute the program giving it a different value.

Press BACK to edit your program; HELP for more help.

C.3 An Example of Static and Dynamic Analysis Being Used In a Sorting Program

PL/I EXECUTION

```
EXAMPL3: .PROCEDURE .OPTIONS (MAIN) ;
.........DECLARE .NUMBERS (5) ;
.........DECLARE .I .FIXED, .J .FIXED, .TEMP;
.........GET .LIST . (NUMBERS) ;
.........DO .I=1 .TO .5;
............DO .J=1 .TO .5;.
..............IF . .NUMBERS (J) >NUMBERS (J+1) .THEN .DO;
.................TEMP=NUMBERS (J) ;
.................NUMBERS (J) =NUMBERS (J+1) ;
.................NUMBERS (J+1) =TEMP;
.................END;
..............END;
............END;
.........PUT .SKIP .LIST (NUMBERS) ;
.........END;
```

Execution Error: FIRST SUBSCRIPT OUT OF RANGE.
Press HELP for error analysis; BACK for list of other options.

(STATIC) ANALYSIS

```
EXAMPL3: .PROCEDURE .OPTIONS (MAIN) ;
.........DECLARE .NUMBERS (5) ;
.........DECLARE .I .FIXED, .J .FIXED, .TEMP;
.........GET .LIST . (NUMBERS) ;
.........DO .I=1 .TO .5;
............DO .J=1 .TO .5;.
...............IF . .NUMBERS (J) >NUMBERS (J+1) .THEN .DO;
..................TEMP=NUMBERS (J) ;
..................NUMBERS (J) =NUMBERS (J+1) ;
..................NUMBERS (J+1) =TEMP;
..................END;
...............END;
............END;
.........PUT .SKIP .LIST (NUMBERS) ;
.........END;
```

POSITION OF ERROR

You have DECLARE'd the array in question to have the integers from 1 to 5 be legal values for subscript number 1. Thus the array allows 5 possible values for subscript number 1. You have referenced it with a subscript of 6 which is not in this range.
If you change your declaration to look like this:

```
DECLARE      NUMBERS( 1: 6)
```

it will contain 6 possible values for the subscript in question, and your error will be corrected.

Press HELP for more assistance.

EXECUTION-TIME ERROR ANALYSIS

```
EXAMPL3: .PROCEDURE .OPTIONS (MAIN) ;
. . . . . .DECLARE .NUMBERS (5) ;
. . . . . .DECLARE .I .FIXED, .J .FIXED, .TEMP;
. . . . . .GET .LIST . (NUMBERS) ;
. . . . . .DO .I=1 .TO .5;
. . . . . .DO .J=1 .TO .5;
. . . . . .IF . .NUMBERS (J) >NUMBERS (J+1) .THEN .DO;
. . . . . .TEMP=NUMBERS (J) ;
. . . . . .NUMBERS (J) =NUMBERS (J+1) ;
. . . . . .NUMBERS (J+1) =TEMP;
. . . . . .END;
. . . . . .END;
. . . . . .END;
. . . . . .PUT .SKIP .LIST (NUMBERS) ;
. . . . . .END;
                                        POSITION OF ERROR
```

This statement gave J an incorrect value .

Press BACK to edit your program; HELP for more help.

EXECUTION-TIME ERROR ANALYSIS

```
EXAMPL3: .PROCEDURE .OPTIONS (MAIN) ;
. . . . . .DECLARE .NUMBERS (5) ;
. . . . . .DECLARE .I .FIXED, .J .FIXED, .TEMP;
. . . . . .GET .LIST . (NUMBERS) ;
. . . . . .DO .I=1 .TO .5;
. . . . . .DO .J=1 .TO .5;.
. . . . . .IF . .NUMBERS (J) >NUMBERS (J+1) .THEN .DO;
. . . . . .TEMP=NUMBERS (J) ;
. . . . . .NUMBERS (J) =NUMBERS (J+1) ;
. . . . . .NUMBERS (J+1) =TEMP;
. . . . . .END;
. . . . . .END;
. . . . . .END;
. . . . . .PUT .SKIP .LIST (NUMBERS) ;
. . . . . .END;
                                        POSITION OF ERROR
```

This statement gave J an incorrect value .

On the assuption that all your previous responses were correct, the indicated expression is an incorrect constant for the logic of your program.

Press BACK to edit your program; HELP for more help.

EXECUTION-TIME ERROR ANALYSIS

```
EXAMPL3: .PROCEDURE .OPTIONS (MAIN) ;
.........DECLARE .NUMBERS (5) ;
.........DECLARE .I .FIXED, .J .FIXED, TEMP;
.........GET .LIST . (NUMBERS) ;
.........DO .I=1 .TO .5;
..........DO .J=1 .TO .5;.
.............IF . .NUMBERS (J) >NUMBERS (J+1)  THEN DO;
.................TEMP=NUMBERS (J) ;
.................NUMBERS (J) =NUMBERS (J+1) ;
.................NUMBERS (J+1) =TEMP;
.................END;
..............END;
...........END;
.........PUT .SKIP .LIST (NUMBERS) ;
.........END;
                                        POSITION OF ERROR
```

This statement gave J an incorrect value .

On the assuption that all your previous responses were correct, the indicated expression is an incorrect constant for the logic of your program.

No more help available. Press BACK to edit; LAB to rerun.

C.4 A More Complex Example of Dynamic Analysis

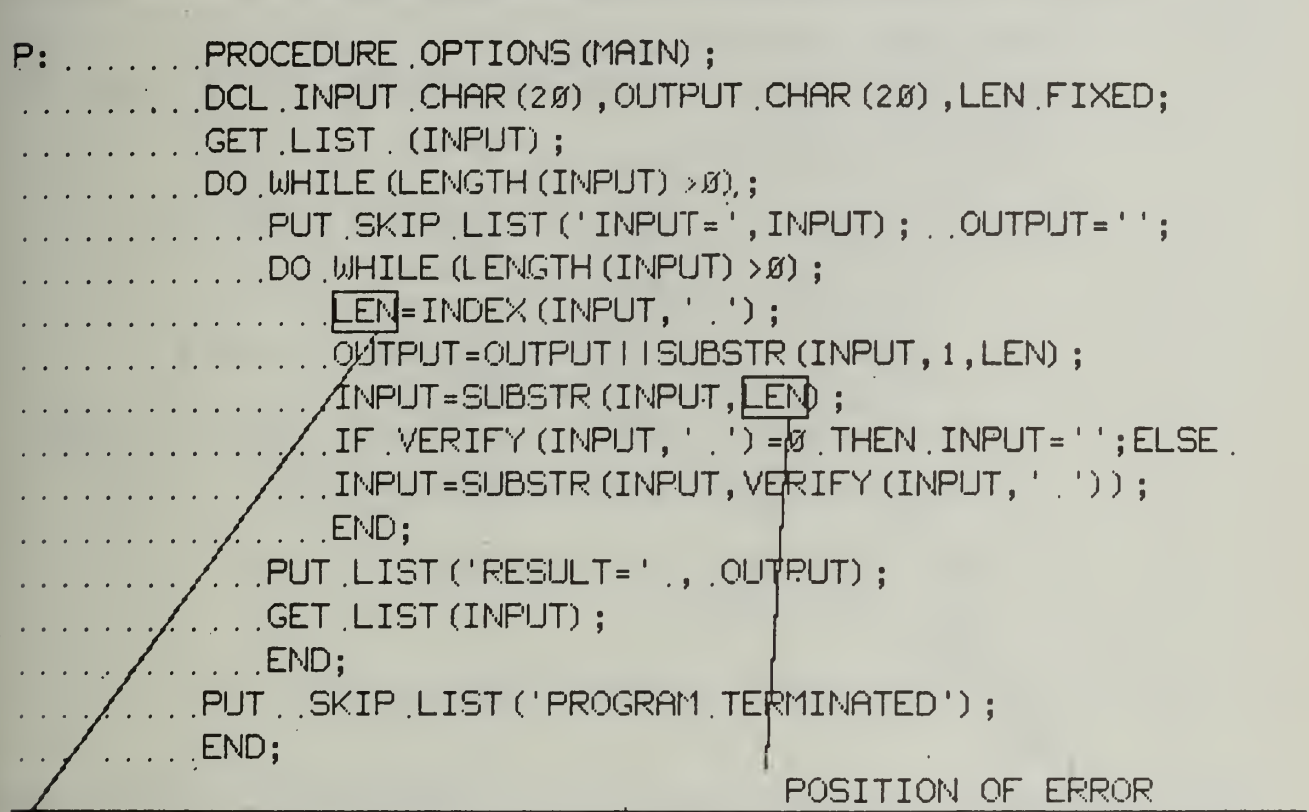
PL/I EXECUTION

```
P: . . . . . .PROCEDURE .OPTIONS (MAIN) ;
. . . . . .DCL .INPUT .CHAR (2Ø) ,OUTPUT .CHAR (2Ø) ,LEN .FIXED;
. . . . . .GET .LIST . (INPUT) ;
. . . . . .DO .WHILE (LENGTH (INPUT) >Ø),;
. . . . . .PUT .SKIP .LIST ('INPUT=' ,INPUT) ; . .OUTPUT='';
. . . . . .DO .WHILE (LENGTH (INPUT) >Ø) ;
. . . . . .LEN=INDEX (INPUT,' ') ;
. . . . . .OUTPUT=OUTPUT||SUBSTR (INPUT, 1 ,LEN) ;
. . . . . .INPUT=SUBSTR (INPUT,LEN) ;
. . . . . .IF .VERIFY (INPUT,' .') =Ø .THEN .INPUT='';ELSE .
. . . . . .INPUT=SUBSTR (INPUT,VERIFY (INPUT,' .')) ;
. . . . . .END;
. . . . . .PUT .LIST ('RESULT=' ., .OUTPUT) ;
. . . . . .GET .LIST (INPUT) ;
. . . . . .END;
. . . . . .PUT . .SKIP .LIST ('PROGRAM .TERMINATED') ;
. . . . . .END;
```

```
INPUT=              THIS . .IS . .WRONG
```

Execution Error: INDEX OF SUBSTRING < 1.
Press HELP for error analysis; BACK for list of other options.

EXECUTION-TIME ERROR ANALYSIS



This statement gave LEN an incorrect value of 0.

Press BACK to edit your program; HELP for more help.
Press DATA to see how execution flowed from here to error.

EXECUTION-TIME ERROR ANALYSIS

```
P: . . . . . .PROCEDURE .OPTIONS (MAIN) ;
. . . . . .DCL .INPUT .CHAR (2Ø) ,OUTPUT .CHAR (2Ø) ,LEN .FIXED;
. . . . . .GET .LIST . (INPUT) ;
. . . . . .DO .WHILE (LENGTH (INPUT) >Ø),;
. . . . . .PUT .SKIP .LIST ('INPUT=' ,INPUT) ; . .OUTPUT='';
. . . . . .DO .WHILE (LENGTH (INPUT) >Ø) ;
. . . . . .LEN=INDEX (INPUT, '.') ;
. . . . . .OUTPUT=OUTPUT||SUBSTR (INPUT, 1 ,LEN) ;
. . . . . .INPUT=SUBSTR (INPUT,LEN) ;
. . . . . .IF .VERIFY (INPUT, '.') =Ø .THEN .INPUT='';ELSE .
. . . . . .INPUT=SUBSTR (INPUT,VERIFY (INPUT, '.')) ;
. . . . . .END;
. . . . . .PUT .LIST ('RESULT=' ., .OUTPUT) ;
. . . . . .GET .LIST (INPUT) ;
. . . . . .END;
. . . . . .PUT . .SKIP .LIST ('PROGRAM .TERMINATED') ;
. . . . . .END;
```

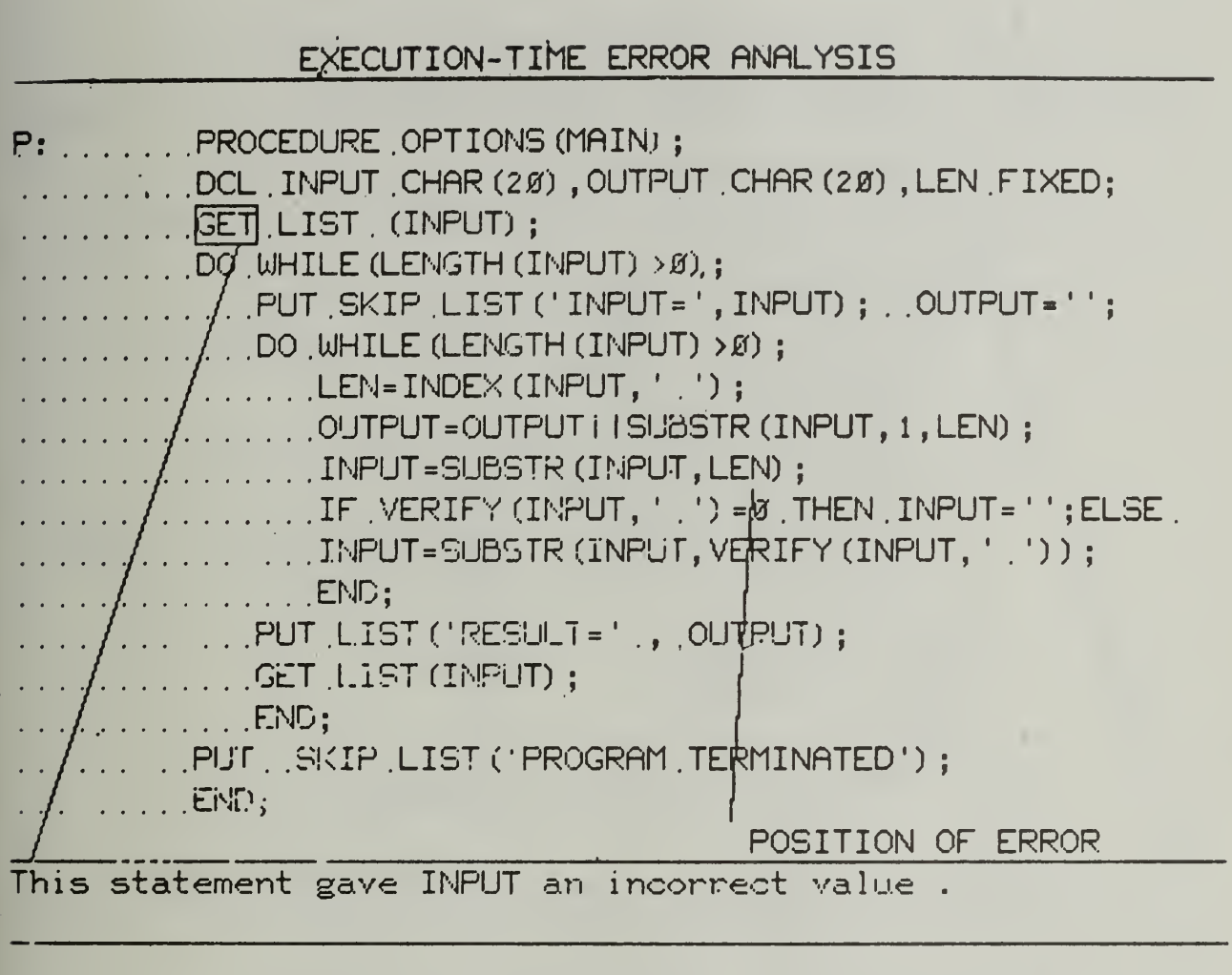
POSITION OF ERROR

This statement gave LEN an incorrect value of Ø.

INDEX returned a value of zero here because the string '.' could not be found anywhere in the string 'WRONG'.

Press BACK to edit your program; HELP for more help.

. . . and eventually:



EXECUTION-TIME ERROR ANALYSIS

```
P: . . . . . .PROCEDURE .OPTIONS (MAIN) ;
. . . . . . . .DCL .INPUT .CHAR (2Ø) ,OUTPUT .CHAR (2Ø) ,LEN .FIXED;
. . . . . . . .GET .LIST . (INPUT) ;
. . . . . . . .DO .WHILE (LENGTH (INPUT) >Ø) ;
. . . . . . . . . .PUT .SKIP .LIST ('INPUT=' ,INPUT) ; . .OUTPUT='' ;
. . . . . . . . . .DO .WHILE (LENGTH (INPUT) >Ø) ;
. . . . . . . . . . . . .LEN=INDEX (INPUT, ' .') ;
. . . . . . . . . . . . .OUTPUT=OUTPUT ! !SUBSTR (INPUT, 1 ,LEN) ;
. . . . . . . . . . . . .INPUT=SUBSTR (INPUT,LEN) ;
. . . . . . . . . . . . .IF .VERIFY (INPUT, ' .') =Ø .THEN .INPUT='' ;ELSE .
. . . . . . . . . . . . . . .INPUT=SUBSTR (INPUT, VERIFY (INPUT, ' .')) ;
. . . . . . . . . . . . .END;
. . . . . . . . . . .PUT .LIST ('RESULT=' ., .OUTPUT) ;
. . . . . . . . . . .GET .LIST (INPUT) ;
. . . . . . . . . . .END;
. . . . . .PUT . .SKIP .LIST ('PROGRAM .TERMINATED') ;
. . . . . .END;
                                                    POSITION OF ERROR
```

This statement gave INPUT an incorrect value .

The value input here for the variable mentioned above is incorrect. Re-execute the program giving it a different value.

Press BACK to edit your program; HELP for more help.

VITA

Alan Davis was born in New York City on January 6, 1949. He graduated from the State University of New York at Albany with a Bachelor of Science degree in mathematics in June, 1970. In January, 1973, he received a Master of Science degree in computer science from the University of Illinois at Urbana-Champaign. He is a member of the Association of Computing Machinery.

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-75-695	2.	3. Recipient's Accession No.

4. Title and Subtitle
AN INTERACTIVE ANALYSIS SYSTEM FOR EXECUTION-TIME ERRORS

5. Report Date
January 9, 1975

6.

7. Author(s)
Alan M. Davis

8. Performing Organization Rept. No.

9. Performing Organization Name and Address
Department of Computer Science
University of Illinois
Urbana, Illinois 61801

10. Project/Task/Work Unit No.

11. Contract/Grant No.
NSF-EC-41511

12. Sponsoring Organization Name and Address
National Science Foundation
Washinton, D.C.

13. Type of Report & Period Covered
Doctoral - 1975

14.

15. Supplementary Notes

16. Abstracts

The interactive analysis system for execution-time errors is designed for use in introductory computer programming courses. Its purpose is to provide the facility to have execution-time errors in higher-level programs automatically analyzed so as to find the actual cause for the errors. It is written in the TUTOR language of the PLATO computer-assisted instruction system.

17. Key Words and Document Analysis. 17a. Descriptors
Execution-time Errors, Error Correction, Compilers, Debugging Systems, CAI, Computer-Assisted Instruction, Computer Science Education

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report)	21. No. of Pages
UNLIMITED RELEASE	UNCLASSIFIED	107
	20. Security Class (This Page) UNCLASSIFIED	22. Price